

Versioning avec Git

Rémi Boutteau

IUT de Rouen, DUT MMI

Année scolaire 2020/2021

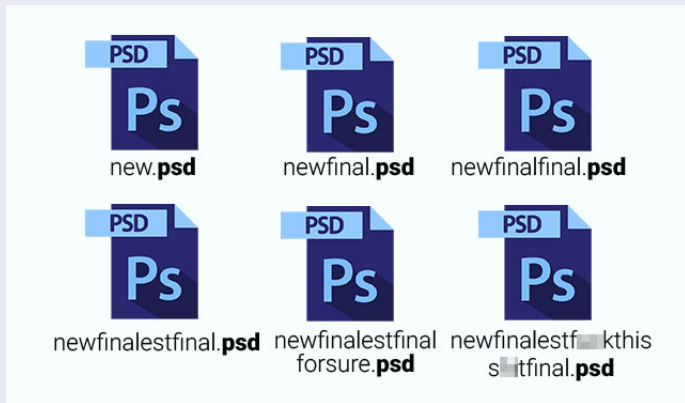
Problématiques

Quel développeur n'a jamais été confronté aux problèmes suivants ?

- ma modification n'a pas marché, et j'ai oublié de faire une sauvegarde de mon code avant...
- qui a touché à mon fichier ? Il y a un bug maintenant...
- ne touche surtout pas à ce fichier, je suis en train de le modifier...
- à quoi servent ces nouveaux fichiers ?
- Aaarg ! un développeur a supprimé des fichiers par mégarde....
- Deux développeurs ont modifié le même fichier en même temps. Comment les fusionner ?

Problématiques

Bref, si vos fichiers ressemblent à ça...



... il est temps de vous intéresser au versioning !

Solution : le versioning !

Le versioning, ou gestion des versions en français, présente de nombreux intérêts :

- historisation des fichiers,
- partage du code source entre plusieurs développeurs,
- gestion des conflits...

Au programme :

- Introduction sur le versioning,
- Créer un dépôt, ajouter des commits, afficher des informations,
- Tagger, créer des branches, merger des branches,
- Annuler ou modifier des commits,
- Travailler avec un dépôt distant,
- Collaborer sur le dépôt de quelqu'un d'autre.

Les principaux systèmes de versioning

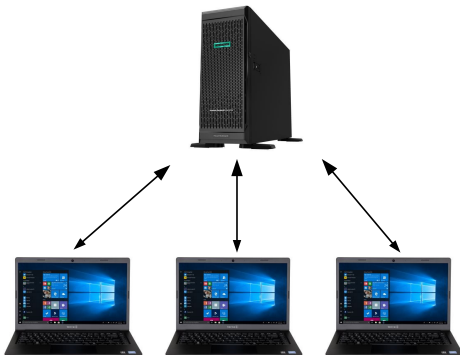
- Git
- Subversion
- Mercurial



Deux types de systèmes de gestion des versions :

- Centralisés : un ordinateur (ou serveur central) qui contient tout le projet,

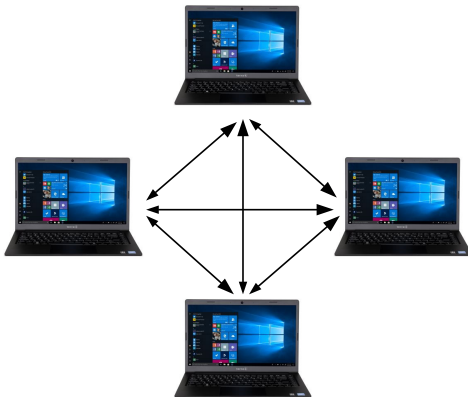
Centralisé



Deux types de systèmes de gestion des versions :

- Centralisés,
- Distribués : pas d'ordinateur central, chaque développeur a la version complète du projet.

Distribué



git

Dans ce cours nous allons étudier git, qui est un système de contrôle de versions distribué.

git est un logiciel libre créé en 2005 par Linus Torvalds, auteur du noyau Linux. Il est multiplateforme.

C'est le logiciel de gestion de versions le plus populaire, utilisé par plus de douze millions de personnes.

Adresse officielle : <https://git-scm.com/>

Origine du nom git

"git" signifie "connard" en argot britannique. Quand on a demandé à Linus Torvalds pourquoi il l'a appelé comme cela, il a répondu : "je ne suis qu'un sale égoцентриque, donc j'appelle tous mes projets d'après ma propre personne. D'abord Linux, puis Git".

git vs GitHub



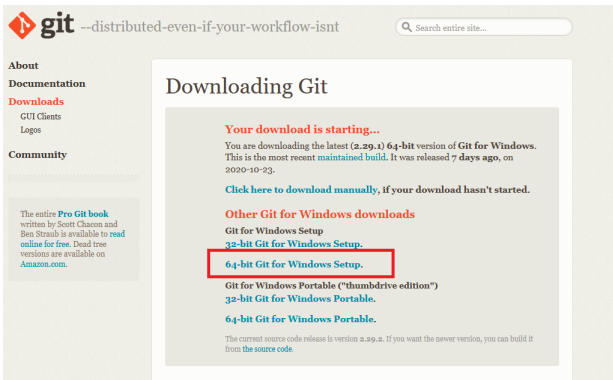
Outil de contrôle des versions



Service qui héberge les projets git

Installation de git

L'installation de git est très simple, il suffit d'aller sur la page "Download" et de télécharger la version qui correspond à votre OS. Il suffit ensuite de lancer l'installation en laissant tous les paramètres par défaut.



The screenshot shows the Git website's 'Downloading Git' page. The header features the Git logo and the tagline '--distributed-even-if-your-workflow-isnt', along with a search bar. The left sidebar contains navigation links: 'About', 'Documentation', 'Downloads' (highlighted in red), 'GUI Clients', 'Logos', and 'Community'. Below these is a text block about the 'Pro Git' book. The main content area is titled 'Downloading Git' and includes the following sections:

- Your download is starting...**: A message stating that the latest (2.29.1) 64-bit version of Git for Windows is being downloaded, released 7 days ago (2020-10-23). It includes a link to 'Click here to download manually, if your download hasn't started.'
- Other Git for Windows downloads**: A section listing various download options:
 - Git for Windows Setup
 - 32-bit Git for Windows Setup.
 - 64-bit Git for Windows Setup.** (This link is highlighted with a red rectangular box)
 - Git for Windows Portable ("thumbdrive edition")
 - 32-bit Git for Windows Portable.
 - 64-bit Git for Windows Portable.
- A footer note: 'The current source code release is version 2.29.2. If you want the newer version, you can build it from the source code.'

Vérification de l'installation de Git

Après l'installation de git, vous pouvez lancer un terminal classique ou lancer "Git CMD" depuis le menu Démarrer. Pour vérifier que git a bien été installé, nous allons exécuter la commande suivante :

```
git --version
```

Si git est bien installé sur votre machine, vous obtiendrez le numéro de version :

```
C:\Users\boutt>git --version  
git version 2.28.0.windows.1
```

Configuration de git

La première fois qu'on utilise git, il faut faire quelques configurations :

```
# Configuration du nom
git config --global user.name "<Votre Nom Complet>"

# Configuration de l'email
git config --global user.email "<Votre email>"

# Configuration de la couleur dans les sorties de
git
git config --global color.ui auto

# Pour voir la Configuration
git config --list
```

Configuration de git

Il est également possible de définir quel éditeur de code on souhaite utiliser avec git. Voici quelques exemples d'éditeurs courants, pour les autres, on trouvera la ligne de commande en faisant une recherche sur internet.

Sublime Text :

```
git config --global core.editor "'C:/Program Files/  
Sublime Text 2/sublime_text.exe' -n -w"
```

Visual Studio Code :

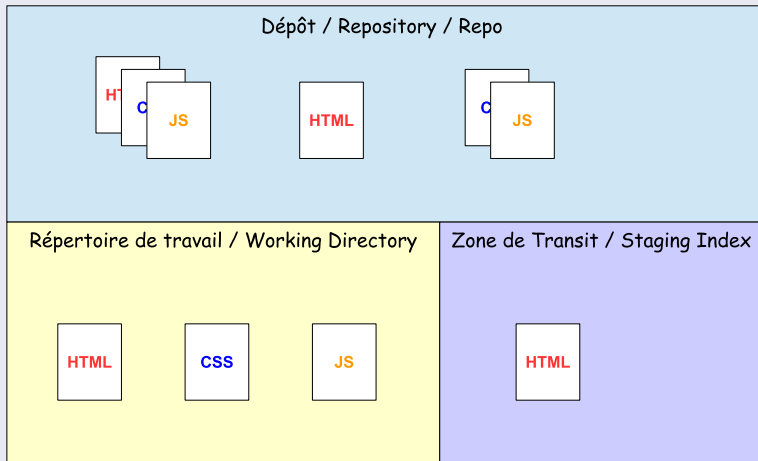
```
git config --global core.editor "code --wait"
```

Notepad++ :

```
git config --global core.editor "'C:/Program Files  
(x86)/Notepad++/notepad++.exe' -multiInst -  
notabbar -nosession -noPlugin"
```

Fonctionnement général de git

Il y a trois zones importantes :



Fonctionnement général de git

Il y a trois zones importantes :

- le répertoire de travail (working directory) : c'est le dossier dans lequel vous êtes en train de travailler et qui contient la dernière version de vos fichiers,
- le dépôt (repository / repo) : c'est lui qui contient tout l'historique de vos fichiers,
- la zone de transit (staging index) : on ne peut pas envoyer les fichiers directement du répertoire de travail au dépôt, il faut passer par une zone intermédiaire, la zone de transit. Cette zone sert donc à définir quels changements vont effectivement être envoyés dans le dépôt.

Premier dépôt local sous git

Pour créer un dépôt vide, il faut d'abord créer un dossier vide qui contiendra votre projet.

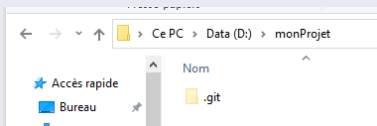
Une fois ce dossier créé, il faut se déplacer à l'intérieur de celui-ci puis initialiser un dépôt git grâce à la commande :

```
git init
```

Vous devriez obtenir :

```
D:/monProjet>git init
Initialized empty Git repository in D:/monProjet/.git/
```

Et un dossier caché a été créé :



git status

Nous avons maintenant un dépôt local. Avec la commande "git status", nous pouvons voir le statut du dépôt :

```
D:/monProjet>git status
```

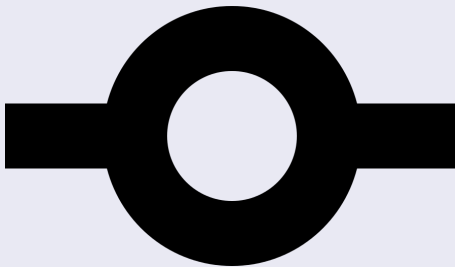
```
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git  
add" to track
```

On voit qu'il n'y a pas encore de "commit" ni de fichiers à "commiter".

Ajouter des commits à un dépôt



Terminologie : commit

Un commit est comme une photographie des fichiers à un moment donné. A chaque fois qu'on fera un commit, on copiera l'état des fichiers dans le dépôt. C'est un peu comme la fonction "Sauvegarder" d'un jeu vidéo ou d'un logiciel. Le commit est l'opération fondamentale dans git.

Premier commit

Nous allons créer un projet "standard" pour un site internet :

- Créez un fichier index.html
- Créez un dossier css. Dans ce dossier, créez un fichier app.css
- Créez un dossier js. Dans ce dossier, créez un fichier app.js
- Complétez index.html avec le minimum vital et en faisant référence aux fichiers app.css et app.js. Ces deux fichiers peuvent être vides pour le moment.

Si nous refaisons un git status, nous avons maintenant :

```
D:\monProjet>git status
On branch master

No commits yet

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

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

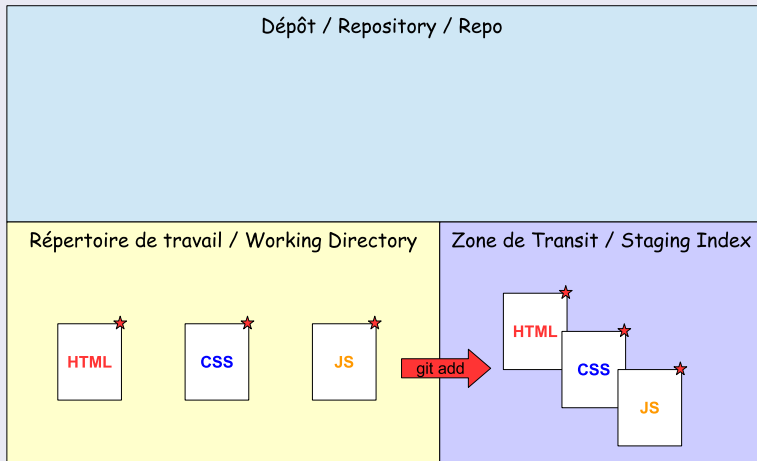
Premier commit

Voilà la situation actuelle :

- nous avons 3 nouveaux fichiers que nous voulons suivre : index.html, app.css dans le dossier css et app.js dans le dossier js.
- pour que Git suive un fichier, il doit être "commité" dans le dépôt
- pour être "commités", les fichiers doivent être dans la zone de transit.

La première étape va donc être d'ajouter ces fichiers dans la zone de transit en utilisant la commande git add.

Premier commit : situation globale



Premier commit

Nous allons commencer par le fichier index.html. Nous ajouterons les autres ultérieurement :

```
git add index.html
```

Un git status nous donnera alors :

```
D:\monProjet>git status
On branch master

No commits yet

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

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


Premier commit

Nous avons maintenant une section "Changes to be committed". Cette section contient les fichiers qui sont dans la zone de transit (staging area). Pour l'instant il n'y a que le fichier index.html. Si nous faisons le commit maintenant, seul ce fichier sera envoyé dans le dépôt.

Dans la console il est également indiqué :

```
use "git rm --cached <file>..." to unstage
```

Cette commande sert à retirer un fichier qu'on aurait mis accidentellement dans la zone de transit. Cette commande le retire de la zone de transit, mais elle ne supprime pas le fichier dans le répertoire de travail.

Premier commit

Il nous reste à envoyer les autres fichiers dans la zone de transit, on peut le faire de la façon suivante :

```
git add css/app.css js/app.js
```

Ici nous n'avons que trois fichiers, mais si on avait des dizaines de fichier, cela serait fastidieux !

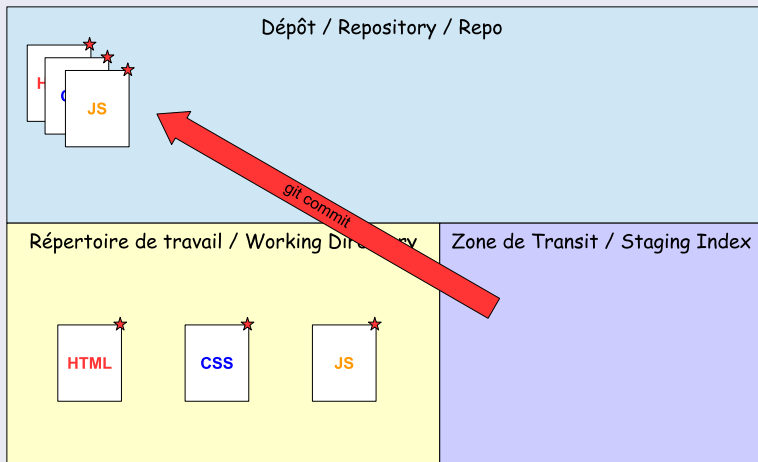
Pour aller plus vite, on peut utiliser le "." qui signifie "ajouter tous les dossiers et fichiers du répertoire courant". On peut donc ajouter tous nos fichiers par :

```
git add .
```

Vérifiez que cela a marché avec un `git status`.

Premier commit

Nous allons maintenant utiliser la commande "commit" pour transférer les fichiers de la zone de transit vers le dépôt.

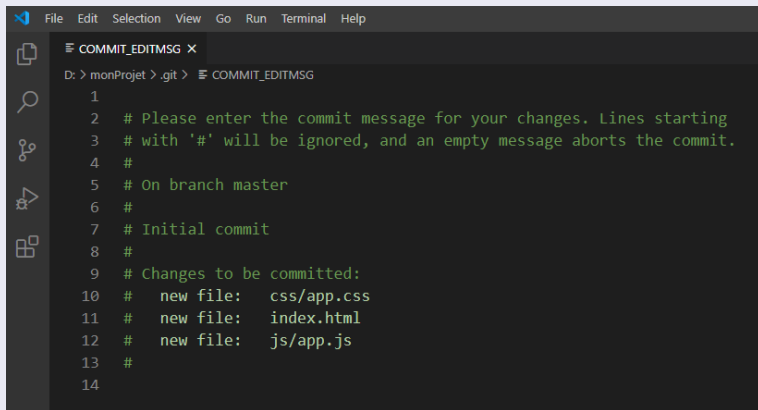


Premier commit

La commande :

```
git commit
```

lance l'éditeur de code que vous avez configuré :

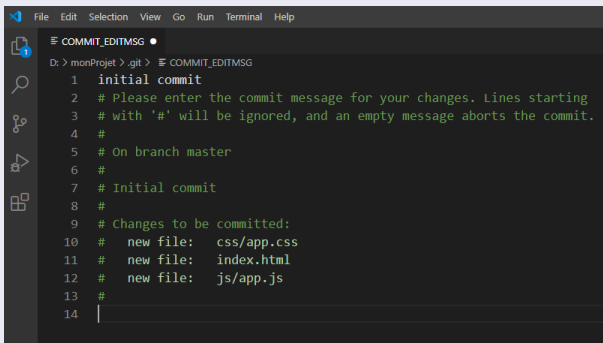
A screenshot of a code editor window titled "COMMIT_EDITMSG X". The editor shows the standard Git commit message template. The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The left sidebar contains icons for Explorer, Search, Source Control, Run and Debug, and Extensions. The main text area contains the following content:

```
D: > monProjet > .git > COMMIT_EDITMSG
1
2  # Please enter the commit message for your changes. Lines starting
3  # with '#' will be ignored, and an empty message aborts the commit.
4  #
5  # On branch master
6  #
7  # Initial commit
8  #
9  # Changes to be committed:
10 #   new file:   css/app.css
11 #   new file:   index.html
12 #   new file:   js/app.js
13 #
14
```

Premier commit

Comme l'indique le message, il faut entrer le message du commit et tout ce qui est précédé par `#` sera ignoré.

Comme c'est la première fois qu'on "commite" les fichiers du projet, nous allons mettre comme message "initial commit" :



```
COMMIT_EDITMSG
D: > monProjet > .git > COMMIT_EDITMSG
1  initial commit
2  # Please enter the commit message for your changes. Lines starting
3  # with '#' will be ignored, and an empty message aborts the commit.
4  #
5  # On branch master
6  #
7  # Initial commit
8  #
9  # Changes to be committed:
10 #   new file:   css/app.css
11 #   new file:   index.html
12 #   new file:   js/app.js
13 #
14 |
```

Il suffit ensuite de fermer la fenêtre pour que le commit se termine.

Premier commit

Bravo, vous avez réussi votre premier commit !

On peut le vérifier grâce à la commande `git status` :

```
D:\monProjet>git status
On branch master
nothing to commit, working tree clean
```

Astuce : Dans le cas d'un message court, on peut se passer de l'éditeur en entrant directement le message grâce à la commande :

```
git commit -m "Initial commit"
```

Deuxième commit

Modifions, un de nos fichiers, index.html en lui ajoutant un header :

```
<header>  
  <h1>Accueil</h1>  
</header>
```

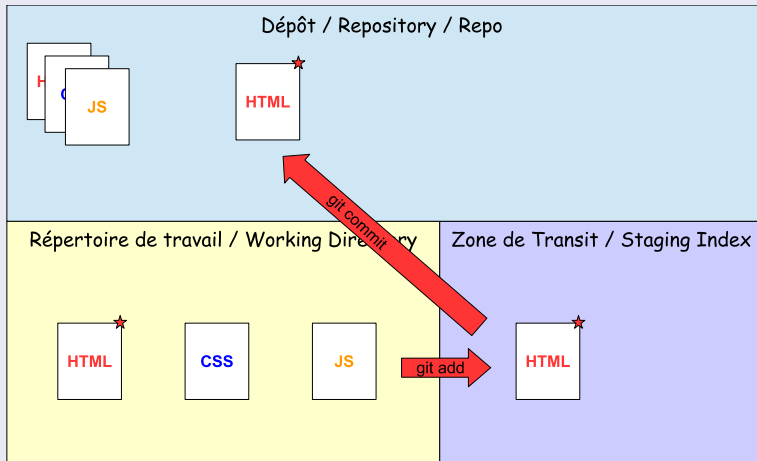
Si on exécute de nouveau git status :

```
D:\monProjet>git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git restore <file>..." to discard changes in working directory)  
        modified:   index.html  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

Git a bien détecté que le fichier index.html a été modifié. Si on veut de nouveau commiter ce message, il faudra :

```
git add index.html # ou git add . pour le mettre  
dans la zone de transit  
git commit -m "Ajout d'un header a notre site"
```

Deuxième commit



Quizz

Je veux changer la couleur de certains titres d'articles de mon site.
J'ai donc :

- modifié le fichier HTML pour donner une classe à ces titres,
- modifié le fichier CSS pour ajouter cette nouvelle classe et donner une couleur,
- enregistré tous mes fichiers,
- lancé git commit dans la console.

Que se passe-t-il ?

Quand doit-on faire un commit ?

Chaque commit ne doit avoir qu'**un seul but**. C'est parfois assez subjectif, mais un commit ne doit changer qu'un aspect du projet. Ca n'est pas lié au nombre de lignes de code changées ou fichier ajoutés, par exemple, si on veut ajouter une image on va généralement :

- ajouter l'image dans les fichiers du projet,
- modifier le HTML pour inclure cette image,
- modifier le CSS pour formater l'aspect de cette image (centrage, marges, etc).

Il est donc tout à fait normal de ne faire qu'un commit pour tout cela puisqu'il y a un unique but : inclure l'image.

Quand doit-on faire un commit ?

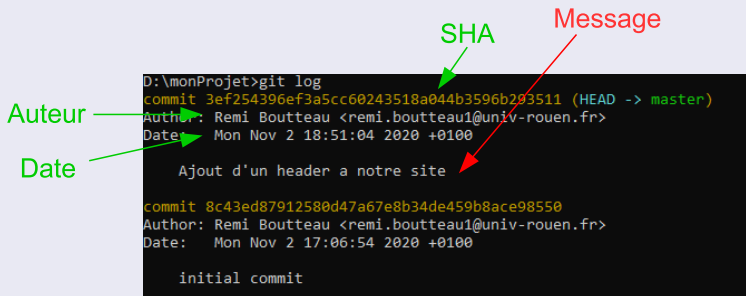
Par contre, un commit ne doit pas inclure des changements qui ne sont pas liés ; par exemple, modifier le menu latéral et changer le texte dans un footer. Dans ce cas, on fera deux commits.

Cela permet, si on doit annuler des changements suite à la découverte d'un bug, de ne pas annuler aussi le deuxième élément. Il faut toujours se poser la question : si les changements introduits par le commit étaient annulés, qu'est-ce qu'il se passerait ? Un commit effacé ne devrait annuler qu'une seule chose.

git log

On peut voir l'historique des commits grâce à la commande :

```
git log
```



Le SHA (Secure Hash Algorithm) est un identifiant unique pour chaque commit calculé automatiquement.

git log

Pour se déplacer dans les logs (dans le cas où ça ne tient pas sur une seule fenêtre) :

- flèche du bas (↓) pour descendre dans les logs,
- flèche du haut (↑) pour remonter dans les logs,
- touche "q" pour quitter et pouvoir de nouveau entrer des commandes dans la console.

git log

Quand il y a de nombreux commits, cet affichage est assez fastidieux à parcourir. De plus, certaines informations ne sont pas vitales (date, adresse mail, affichage complet du SHA, etc). Pour avoir une visualisation plus compacte, on peut utiliser le commande :

```
git log --oneline
```

```
D:\monProjet>git log --oneline
3ef2543 (HEAD -> master) Ajout d'un header a notre site
8c43ed8 initial commit
```

git log

On peut également afficher tous les fichiers qui ont été modifiés dans un commit, le nombre de lignes ajoutées ou supprimées, grâce à la commande `git log --stat` (pour "statistics") :

```
git log --stat
```

```
D:\monProjet>git log --stat
commit 3ef254396ef3a5cc60243518a044b3596b293511 (HEAD -> master)
Author: Remi Boutteau <remi.boutteau1@univ-rouen.fr>
Date:   Mon Nov 2 18:51:04 2020 +0100

    Ajout d'un header a notre site

index.html | 3 +++
1 file changed, 3 insertions(+)

commit 8c43ed87912580d47a67e8b34de459b8ace98550
Author: Remi Boutteau <remi.boutteau1@univ-rouen.fr>
Date:   Mon Nov 2 17:06:54 2020 +0100

    initial commit

css/app.css | 0
index.html | 13 ++++++++
js/app.js   | 0
3 files changed, 13 insertions(+)
```

git log

On peut aller plus loin de le niveau de détails à afficher, notamment en affichant les différences dans les fichiers entre deux commits. Pour cela on utilise la commande `git log -p` (pour "patch") :

```
git log -p
```

```
D:\monProjet>git log -p
commit 3ef254396ef3a5cc60243518a044b3596b293511 (HEAD -> master)
Author: Remi Boutteau <remi.boutteau1@univ-rouen.fr>
Date:   Mon Nov 2 18:51:04 2020 +0100

    Ajout d'un header a notre site

diff --git a/index.html b/index.html
index fc4c025..0a20612 100644
--- a/index.html
+++ b/index.html
@@ -6,6 +6,9 @@
     <link rel="stylesheet" href="css/app.css">
   </head>
   <body>
+   <header>
+     <h1>Accueil</h1>
+   </header>
   <p>Mon projet</p>

   <script src="js/app.js"></script>
```


git show

Pour éviter d'avoir à scroller à la recherche d'un commit spécifique, on peut l'atteindre grâce à la commande `git show` et en spécifiant le SHA :

```
git show 3ef2543
```

```
D:\monProjet>git show 3ef2543
commit 3ef254396ef3a5cc60243518a044b3596b293511 (HEAD -> master)
Author: Remi Boutteau <remi.boutteau1@univ-rouen.fr>
Date: Mon Nov 2 18:51:04 2020 +0100
```

Ajout d'un header a notre site

```
diff --git a/index.html b/index.html
index fc4c025..0a20612 100644
--- a/index.html
+++ b/index.html
@@ -6,6 +6,9 @@
     <link rel="stylesheet" href="css/app.css">
 </head>
 <body>
+ <header>
+   <h1>Accueil</h1>
+ </header>
   <p>Mon projet</p>

   <script src="js/app.js"></script>
```

Un bon message de commit

Le message de commit est sans doute l'information la plus importante. Il est donc essentiel de rédiger de bons messages de commits. Voici quelques instructions à respecter :

A faire :

- écrire un message court (moins de 60 caractères)
- expliquer ce que fait le commit (pas comment ni pourquoi)

A ne pas faire :

- ne pas expliquer pourquoi le changement a été fait
- ne pas expliquer comment le changement a été fait (c'est le but du `git log -p` !)
- ne pas utiliser le mot "et" : si vous utilisez le mot "et", c'est que votre commit fait probablement trop de changements, il faut séparer le commit en plusieurs commits.

Un bon message de commit

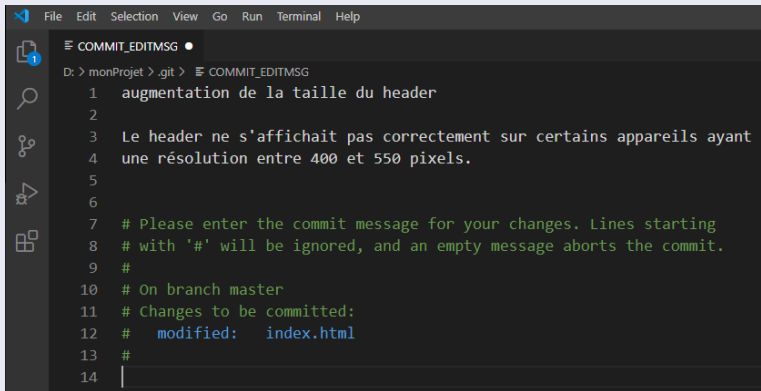
Les messages suivants sont-ils de bons messages de commit ?

- "Mise à jour du footer avec les informations de Copyright"
- "Ajout d'un tag au body"
- "Centrage du logo dans le header"
- "Ajout de changements dans app.js"
- "Modification de la police de caractères des paragraphes"

Expliquer le pourquoi

Si vous voulez expliquer pourquoi un commit a été fait, vous le pouvez.

Quand vous éditez le message de commit, la première ligne est le message. Laissez une ligne vide puis écrivez l'explication.



```
1  augmentation de la taille du header
2
3  Le header ne s'affichait pas correctement sur certains appareils ayant
4  une résolution entre 400 et 550 pixels.
5
6
7  # Please enter the commit message for your changes. Lines starting
8  # with '#' will be ignored, and an empty message aborts the commit.
9  #
10 # On branch master
11 # Changes to be committed:
12 #   modified:   index.html
13 #
14
```

git diff

Parfois, on travaille sur une fonctionnalité mais on n'a pas le temps de la terminer car il est 2h du matin... Quand on reprend le développement, on ne se rappelle plus forcément sur quoi on travaillait depuis le dernier commit.

Si on fait un `git status`, on va voir quels fichiers ont été modifiés mais pas quels sont les changements. Pour cela, on utilise la commande `git diff` :

```
git diff
```

```
D:\monProjet>git diff
diff --git a/index.html b/index.html
index 0a20612..539e688 100644
--- a/index.html
+++ b/index.html
@@ -8,6 +8,8 @@
<body>
  <header>
    <h1>Accueil</h1>
+    <h1>La boutique</h1>
+    <h1>Contact</hi>
  </header>
<p>Mon projet</p>
```

git ignore

Comme nous l'avons vu, plutôt que d'ajouter tous les fichiers un par un avec `git add`, on utilise généralement `git add .` pour ajouter tous les fichiers dans le répertoire courant et les sous-répertoires.

Or, bien souvent on veut ignorer certains fichiers : fichiers volumineux, librairies (jQuery, Bootstrap...) qui peuvent être retrouvés sur d'autres sites, etc...

Pour cet exemple, nous allons ajouter un fichier "informations.txt" que nous ne voudrions pas versionner.

git status

Pour ignorer des fichiers, nous devons créer un fichier spécial ".gitignore" dans le même dossier que celui qui contient le dossier .git, c'est-à-dire dans le dossier parent. Dans ce fichier, il faut lister les fichiers que nous voulons ignorer, ici "informations.txt".

git ignore

Un git status nous donnera alors :

```
git status
```

```
D:\monProjet>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

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

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

git ignore

Inclure tous les fichiers d'une librairie serait très fastidieux de cette façon ! On va utiliser le concept de "globbing" ^a ^b pour aller plus vite. Voici les principaux à retenir :

```
# ignorer tous les fichiers .png
*.png

# mais n'ignore pas le fichier !home.png malgre la
  commande precedente
!home.png

# ignore le repertoire build a la racine du projet
/build
```

a. [https://en.wikipedia.org/wiki/Glob_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

b. <https://git-scm.com/book/en/v2/>

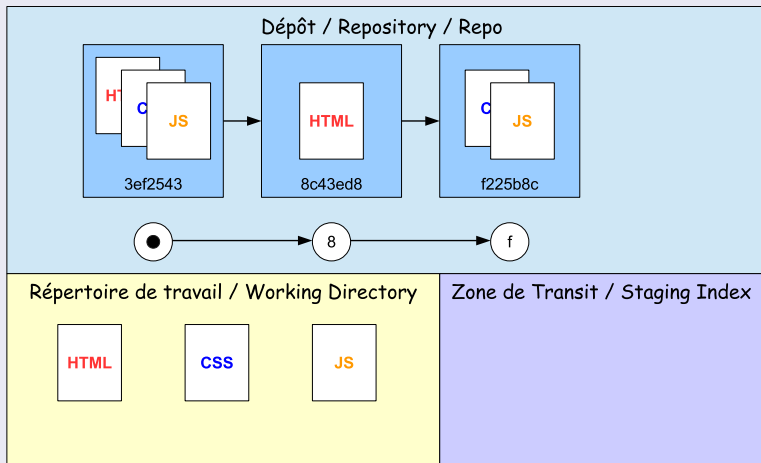
Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files

Tagger, créer des branches, merger des branches



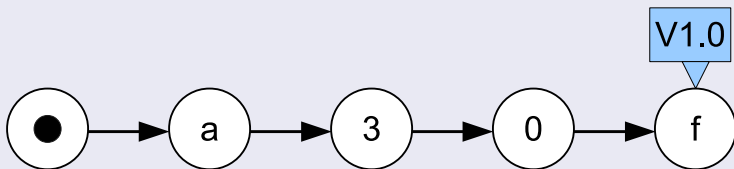
Représentation compacte

Dans la suite, nous utiliserons une représentation plus compacte des commits :



Les tags

Un tag permet d'attacher une annotation à un commit spécifique. Par exemple, supposons que nous venons de terminer une première version de notre site prête à être mise en production, on peut vouloir le signaler en attribuant le tag "v1.0".



Cela sera obtenu grâce à la commande :

```
git tag -a v1.0
```

Les tags

Pour vérifier la liste des tags, dans un dépôt, on utilise :

```
git tag
```

Les tags apparaissent également quand on fait un git log :

```
git log  
git log --oneline
```

```
D:\monProjet>git tag  
v1.0  
  
D:\monProjet>git log --oneline  
a8cb2df (HEAD -> master, tag: v1.0) Ajout des sections Contact et Boutique  
3ef2543 Ajout d'un header a notre site  
8c43ed8 initial commit
```

Les tags

Supprimer un tag :

```
git tag -d v1.0
```

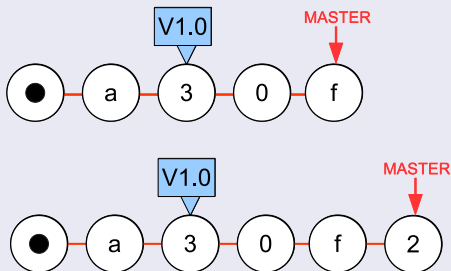
Ajouter un tag à un ancien commit :

```
git tag -a v1.0 3ef2543
```

Les branches

La notion de branche est très importante, c'est une fonction très puissante de git.

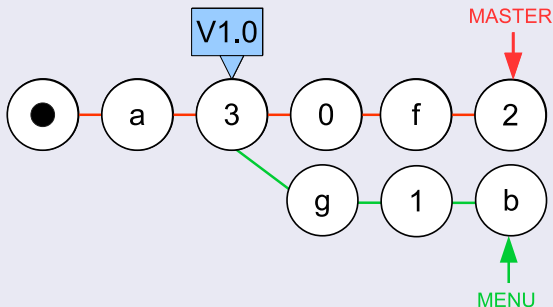
Nous n'en avons pas parlé, mais jusqu'à maintenant, nous n'avions qu'une branche qui est la première créée par git et qui s'appelle "master". A chaque fois qu'on fait un nouveau commit, il est ajouté à cette branche et le pointeur qui pointe vers le dernier commit de la branche se déplace donc d'un commit (contrairement aux tags qui sont fixes).



Les branches

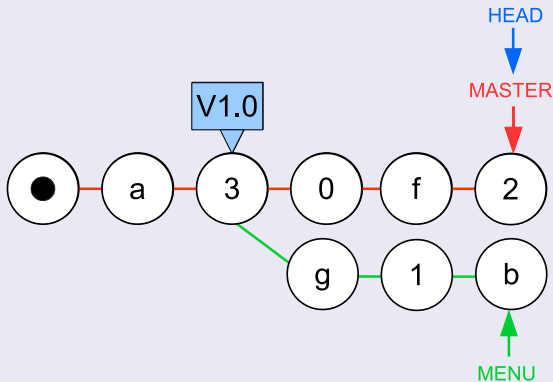
Supposons que nous avons une deuxième branche, par exemple "Menu" dans laquelle nous voulons développer la partie concernant le menu latéral du site.

Si on fait un commit, dans quelle branche sera ajouté le commit ?



Les branches

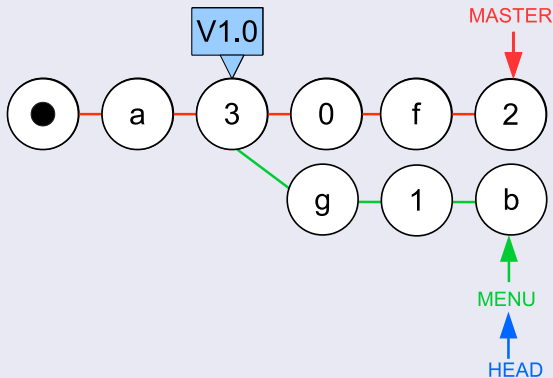
C'est là qu'intervient le pointeur HEAD, qui pointe sur la branche qui est active.



Les branches

Si on veut changer de branche, on utilisera la commande :

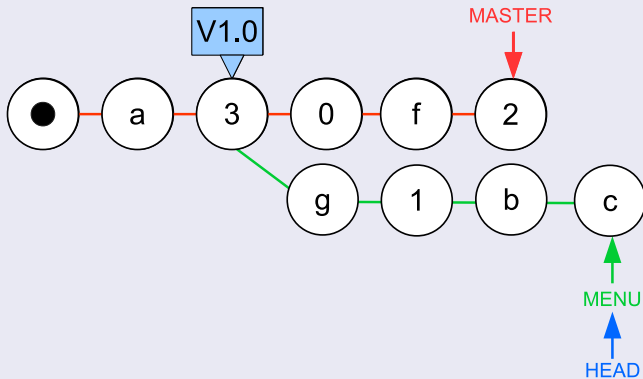
```
git checkout menu
```



Les branches

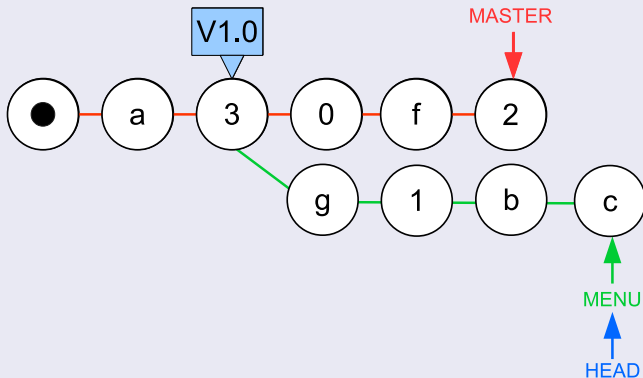
Le commit sera ajouté à la branche active :

```
git checkout menu
```



Les branches

Les fichiers et le code source que l'on verra dans notre éditeur seront ceux de la branche active. Dans l'exemple ci-dessous, on ne verra pas les modifications qui auront été apportées par les commits 0, f et 2 :



git branch

La commande `git branch` permet de voir la liste des branches, ainsi que la branche active (représentée avec le symbole `*`).

```
git branch
```

```
D:\monProjet>git branch
* master
```

Créer une branche

Pour créer une branche appelée "menu", on utilise la commande :

```
git branch menu
```

On peut également créer une branche à partir d'un commit passé en spécifiant son SHA :

```
git branch menu 3ef2543
```

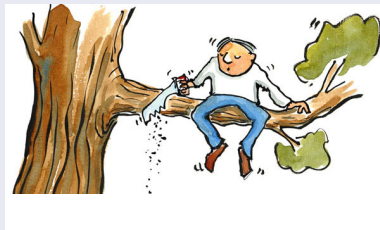
Supprimer une branche

Pour supprimer une branche appelée "menu", on utilise la commande :

```
git branch -d menu
```

Supprimer une branche

Attention : on ne peut pas supprimer la branche sur laquelle on est !



Dans ce cas il faut faire un checkout pour changer de branche avant de la supprimer.

On ne peut pas non plus supprimer une branche dans laquelle il y aurait des commits qui n'auraient pas été mergés (on verra ce que ça veut dire d'ici peu). Si on veut forcer la suppression, il faut utiliser un D majuscule :

```
git branch -D menu
```

Afficher les branches

On peut afficher les branches sous la forme d'un graph avec la commande :

```
git log --oneline --graph --all
```

```
D:\monProjet>git log --oneline --graph --all
* ccb7747 (menu) Ajout du menu
| * a8cb2df (HEAD -> master, tag: v1.0) Ajout des sections Contact et Boutique
|/
* 3ef2543 Ajout d'un header a notre site
* 8c43ed8 initial commit
```

ungit

Pour une meilleure visualisation, on peut utiliser un logiciel tel que ungit :

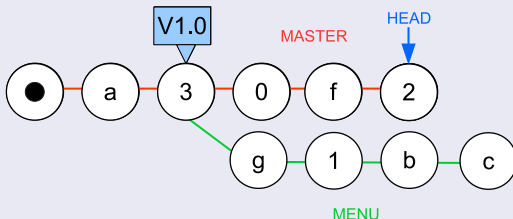
```
npm install -g ungit
ungit
```

Fusionner des branches

Lorsqu'on a terminé le développement d'une fonctionnalité dans une branche, on veut intégrer les modifications dans une autre branche, généralement la branche master. Cela s'appelle fusionner ou "merge" en anglais.

Il faut commencer par se placer sur la branche dans laquelle on veut fusionner :

```
git checkout master
```

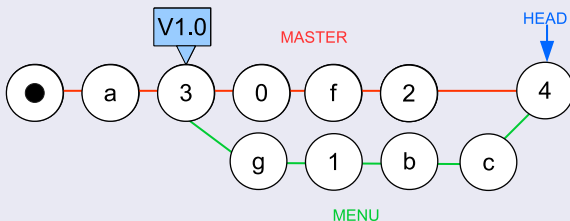


Fusionner des branches

On utilise alors la commande `git merge` en précisant le nom de la branche à fusionner :

```
git merge menu
```

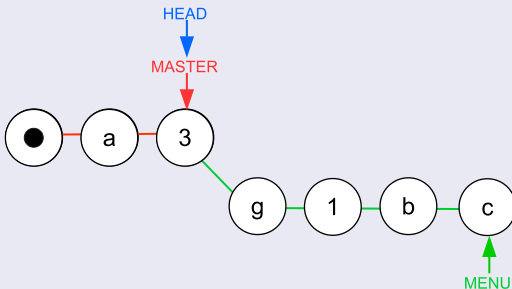
git crée alors un nouveau commit qui contient la fusion des deux branches :



Remarque : une fois la fusion terminée, on peut supprimer la branche qui ne servira plus.

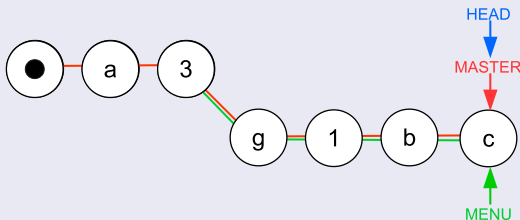
Fast-forward merge

Le fast-forward merge a lieu lorsque le chemin entre les deux branches est linéaire :



Fast-forward merge

Dans ce cas, git doit simplement déplacer les pointeurs sur le dernier commit :



Remarque 1 : Dans ce cas, il n'y a pas de commit de merge de créé.

Remarque 2 : C'est git qui détermine automatiquement quel algorithme de fusion il va utiliser (la commande est la même).

Si on veut quand même un commit de merge, par exemple à des fins d'archivage, on peut exécuter la commande :

```
git merge --no-ff menu
```

Les conflits

La plupart du temps, git est capable de fusionner des branches sans problèmes, même quand il y a eu de nombreuses modifications. Cependant, il y a des cas où la fusion ne peut pas être faite automatiquement. Quand un merge échoue, on appelle cela un **conflit**.

Quand un conflit arrive, git essaie de combiner le maximum de code possible, mais il laissera des caractères spéciaux (<<< et >>>) pour vous dire où vous devez résoudre ces conflits manuellement. git suit les lignes dans les fichiers. Si vous modifiez exactement la même ligne dans deux branches différentes, cela créera un conflit. Par exemple, si vous changez un même titre dans les deux branches. On a alors le message suivant :

```
D:\monProjet>git merge menu
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Les conflits

Git ouvre alors l'éditeur de code pour nous demander de faire manuellement les modifications qui sont indiquées :

```
d: > monProjet > index.html > html > body > div#menu > ?
1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Mon Projet</title>
6   <link rel="stylesheet" href="css/app.css">
7 </head>
8 <body>
9   <header>
10    <h1>Accueil</h1>
11    <h1>La boutique</h1>
12    <h1>Contact</h1>
13  </header>
14  <p>Mon projet</p>
15  <div id="menu">
16    <a href="accueil.html" target="Blank">Accueil</a>
17    <a href="boutique.html" target="Blank">Boutique</a>
18  <<<<<< HEAD (Current Change)
19    <a href="contact" target="Blank">Contact</a>
20    <a href="aboutme.html" target="Blank">A propos de moi</a>
21  =====
22    <a href="contact.html" target="Blank">Contact</a>
23    <a href="aboutme.html" target="Blank">A propos</a>
24  >>>>>> menu (Incoming Change)
25
26  </div>
27
28  <script src="js/app.js"></script>
29 </body>
30 </html>
```

Les conflits

Il faut alors déterminer quelles lignes ont veu garder, supprimer tout ce qui doit être supprimé (y compris les indicateurs) et faire un commit.

Attention : parcourez bien le code, car il peut y avoir des conflits à plusieurs endroits. il est conseiller de faire une recherche sur les caractères <<< pour s'en assurer.

```
d:\monProjet > index.html > html > body > div#menu > ?
1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Mon Projet</title>
6   <link rel="stylesheet" href="css/app.css">
7 </head>
8 <body>
9   <header>
10     <h1>Accueil</h1>
11     <h1>La boutique</h1>
12     <h1>Contact</h1>
13   </header>
14   <p>Mon projet</p>
15   <div id="menu">
16     <a href="accueil.html" target="Blank">Accueil</a>
17     <a href="boutique.html" target="Blank">Boutique</a>
18 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
19 <<<<<< HEAD (Current Change)
20     <a href="contact.html" target="Blank">Contact</a>
21     <a href="aboutme.html" target="Blank">A propos de moi</a>
22 =====
23     <a href="contact.html" target="Blank">Contact</a>
24     <a href="aboutme.html" target="Blank">A propos</a>
25 >>>>>> menu (Incoming Change)
26 </div>
```

Annuler ou modifier des commits



Modifier le dernier commit

Il est possible de modifier le dernier commit grâce à la commande :

```
git commit --amend
```

Il est également possible de modifier des fichiers ou d'en ajouter.
Dans ce cas il faut :

- éditer le fichier
- sauvegarder le fichier
- mettre le fichier dans la zone de transit
- lancer la commande :

```
git commit --amend
```


git revert

Il est possible d'annuler les effets d'un commit spécifique. Cela signifie qu'on va faire exactement l'inverse du commit :

- si des caractères ou des lignes ont été ajoutés, ils vont être supprimés,
- au contraire, des caractères ou lignes supprimées vont réapparaître, etc...

Pour cela, on utilise la commande `git revert` suivie du SHA du commit à annuler : grâce à la commande :

```
git revert b465f4b
```

git reset : dangerous zone

Revert vs Reset : Un git revert crée un nouveau commit qui annule les effets d'un précédent commit. un git reset **efface** les commits. Un git reset est obtenu grâce à la commande git reset suivi du SHA du commit à partir duquel il faut supprimer les commits :

```
git reset f38ad4b
```

Cette commande peut être suivie de trois flags :

- - -mixed : les changements retournent dans le répertoire de travail,
- - -soft : les changements retournent dans la zone de transit,
- - -hard : les changements sont totalement supprimés.

Travailler avec un dépôt distant



GitHub

Dépôt local vs dépôt distant

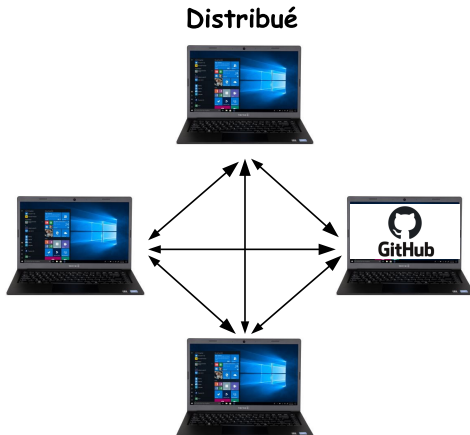
Nous avons vu jusqu'à maintenant l'utilisation de git avec un dépôt local. Cela peut être utile si vous êtes seul à travailler sur un projet. Cependant, il est préférable de créer un dépôt distant pour plusieurs raisons :

- pour pouvoir partager le code entre plusieurs développeurs dans le cas où vous travailler sur un projet commun,
- avoir une copie de votre projet sur un serveur distant, utile en cas de problème sur votre ordinateur (perte de données, vol, dysfonctionnement, etc),
- participer à des projets open-source.

Pour cela, nous allons utiliser GitHub.

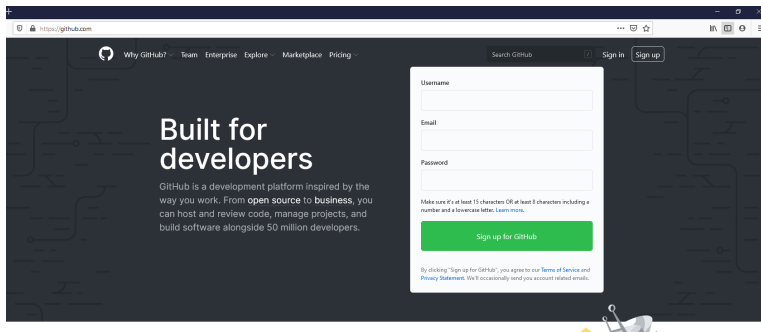
git et GitHub

Pour rappel, git est un système de contrôle de versions **distribué**, c'est-à-dire que chaque développeur a une copie du dépôt. Nous allons donc nous placer dans la configuration suivante :



Création d'un compte GitHub

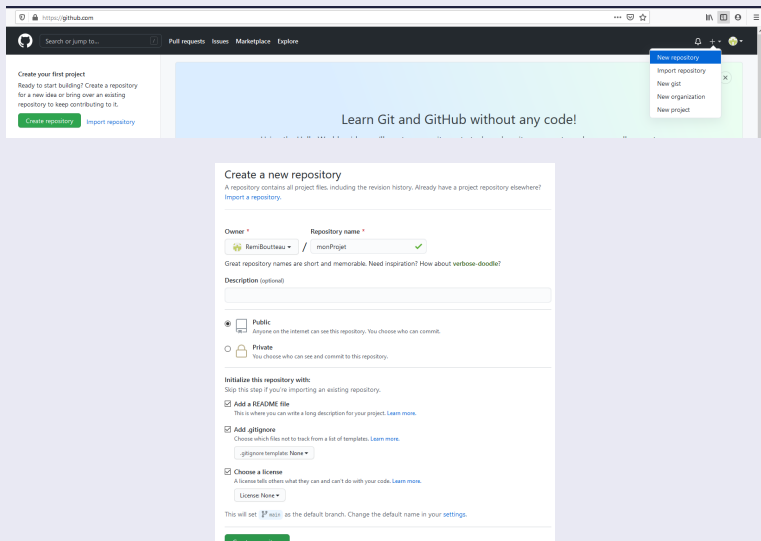
Nous allons donc créer un compte GitHub (gratuit).



Remarque

Jusqu'à récemment (début 2019), il n'était pas possible de créer des dépôts privés avec l'offre gratuite de GitHub. C'est maintenant possible.

Créer un dépôt sur GitHub



https://github.com

Search or jump to... Pull requests Issues Marketplace Explore


Create your first project
Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.
[Create repository](#) [Import repository](#)

Learn Git and GitHub without any code!

New repository
Import repository
New git
New organization
New project

Create a new repository

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

Owner *  RemiBoutreau / Repository name * monProjet ✓

Great repository names are short and memorable. Need inspiration? [How about verbose-doodle?](#)

Description (optional)

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

☐ Private
You choose who can see and commit to this repository.

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

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

☒ Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)
- .gitignore template: None

☒ Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)
License: None

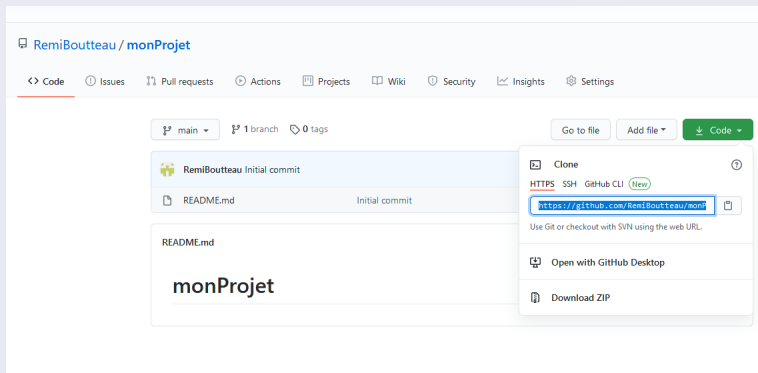
This will set `main` as the default branch. Change the default name in your [settings](#).

[Create repository](#)

Bravo ! Vous avez créé votre premier dépôt.

Accéder à un dépôt distant

Pour accéder à un dépôt distant, il faut récupérer son URL. Il faut retourner sur le dépôt sur GitHub et cliquer sur "Code". Faites alors un "Copier" de l'URL HTTPS du dépôt.



Accéder à un dépôt distant

Retournez dans la console, accédez au dépôt local que vous avez créé, puis tapez la commande suivante (avec votre URL) :

```
git remote add origin https://github.com/  
RemiBoutteau/monProjet.git
```

Remarque : Ce que nous avons fait précédemment ne permet pas de cloner le dépôt, mais juste de pointer vers le dépôt distant.

Vérifier le dépôt distant

On peut vérifier qu'on pointe bien vers un dépôt distant grâce à la commande `git remote -v` :

```
D:/monProjet>git remote -v  
origin https://github.com/RemiBoutteau/monProjet.  
git (fetch  
origin https://github.com/RemiBoutteau/monProjet.  
git (push
```

Les noms courts

Dans l'exemple précédent "origin" est un nom court pour faire référence au dépôt distant (plutôt que d'avoir à taper toute l'URL à chaque commande....).

Ce nom court est local à votre dépôt local (celui de votre machine). On aurait pu choisir n'importe quel nom, mais, par convention, on nomme "origin" le dépôt distant principal.

Les deux commandes suivantes permettent donc d'avoir la liste des dépôts locaux, soit avec leur nom court, soit avec l'adresse complète.

```
git remote  
git remote -v
```

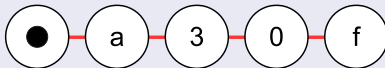
Envoyer des commits sur le dépôt distant

Nous avons pour le moment un dépôt local avec des commits, et un dépôt distant vide pour le moment :

Dépôt Distant

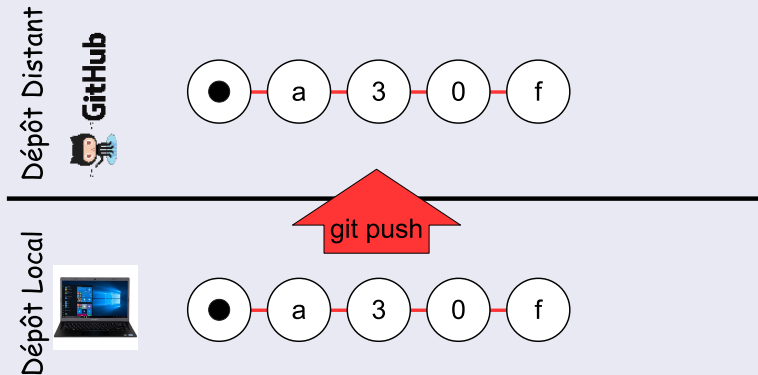


Dépôt Local



git push

La commande git push permet de pousser le dépôt local vers le dépôt distant :



git push

La commande git push doit être suivie du nom du dépôt distant et de la branche qui contient les commits que l'on veut envoyer :

```
git push origin master
```

On obtient alors les informations suivantes qui montrent que l'upload a commencé :

```
D:\monProjet>git push origin master
Enumerating objects: 29, done.
Counting objects: 100% (29/29), done.
Delta compression using up to 8 threads
Compressing objects: 100% (25/25), done.
Writing objects: 100% (29/29), 2.60 KiB | 1.30 MiB/s, done.
Total 29 (delta 13), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (13/13), done.
To https://github.com/RemiBoutteau/monProjet.git
 * [new branch]      master -> master
```

On peut vérifier notre dépôt local :

```
git log --oneline --graph
```

```
D:\monProjet>git log --oneline --graph
* f38ad4b (HEAD -> master, origin/master) Ajout de a propos de moi
|
| * 3825f3b (menu) Ajout de A propos
| * b465f4b Ajout de a propos de moi
|/
* 4f021f8 Merge branch 'menu' into master
|
| * ccb7747 Ajout du menu
| * a8cb2df (tag: v1.0) Ajout des sections Contact et Boutique
|/
* 3ef2543 Ajout d'un header a notre site
* 8c43ed8 initial commit
```

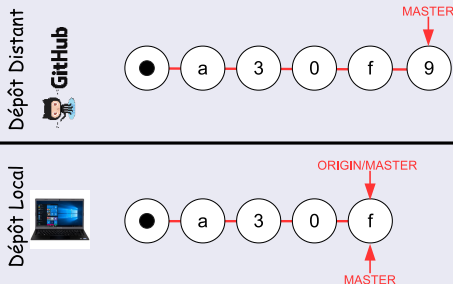
On a un nouvel indicateur origin/master qui est une branche de suivi. Cela nous dit que le dépôt distant origin a une branche master qui pointe sur le commit f38ad4b et qui inclut donc tous les commits précédents. Cela nous permet de suivre les informations du dépôt distant dans notre dépôt local.

Attention : la branche de suivi origin/master n'est pas une représentation en temps-réel de l'état de la branche sur le dépôt distant. Si quelqu'un l'a modifié, elle ne bougera pas dans notre dépôt local.

Changements sur le dépôt distant

Il y a des situations où il peut y avoir des commits sur le dépôt distant que nous n'avons pas dans notre dépôt local. C'est le cas :

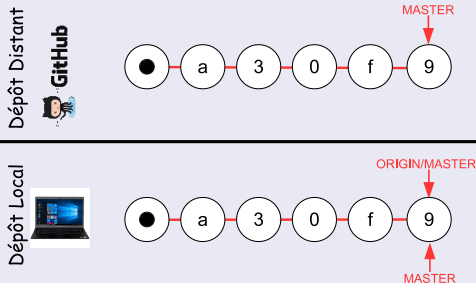
- si vous travaillez dans une équipe et un de vos collègues a poussé des changements sur le dépôt,
- si vous travaillez sur le même projet mais depuis plusieurs ordinateurs, par exemple un ordinateur au travail et un ordinateur personnel.



git pull

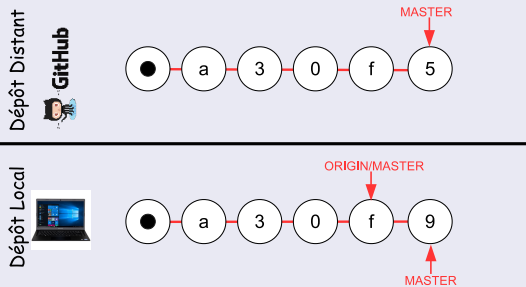
On va alors "tirer" ces commits sur notre dépôt grâce à la commande :

```
git pull origin master
```



git pull

Supposons que nous soyons dans une situation où il y a eu de nouveaux commits sur le dépôt local et sur le dépôt distant :



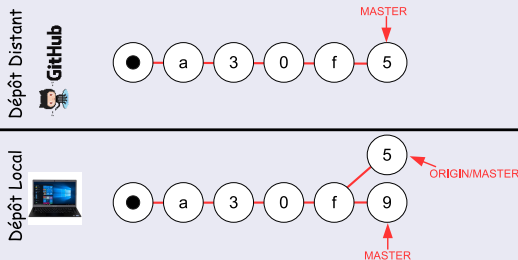
Dans ce cas, la commande `git pull` ne peut pas fonctionner.

pull vs fetch

On va alors utiliser la commande `git fetch`. Cette commande peut-être vue comme un demi git pull :

- les commits du dépôt distant sont copiés dans le dépôt local
- le pointeur de la branche de suivi (origin/master) est bougé pour pointer sur le commit le plus récent,
- le pointeur de la branche master **ne bouge pas**.

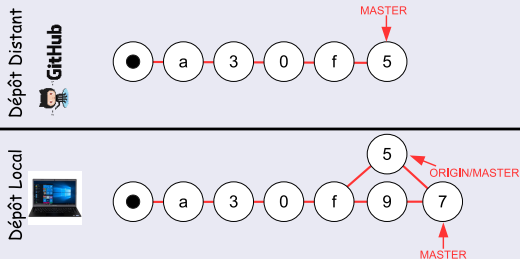
```
git fetch origin master
```



pull vs fetch

Il faudra donc faire un merge pour fusionner les deux commits :

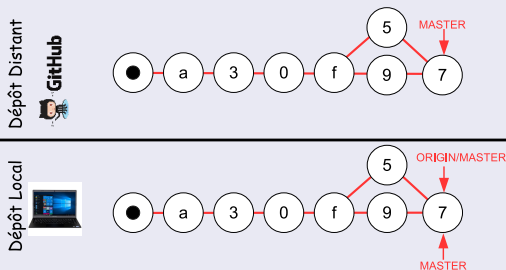
```
git merge origin/master
```



pull vs fetch

Et faire un push pour mettre à jour le dépôt distant :

```
git push origin master
```



Collaborer sur le dépôt de quelqu'un d'autre



Forker un dépôt

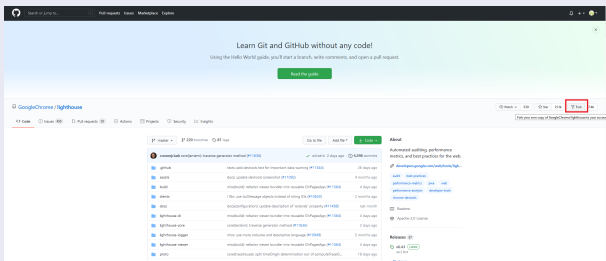
On ne peut pas travailler directement dans le dépôt de quelqu'un d'autre(heureusement!). Il va falloir commencer par dupliquer ce dépôt à l'identique, c'est ce qu'on appelle un "fork".

Forker et cloner un dépôt sont deux choses différentes :

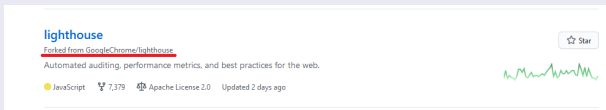
- cloner : on clone un dépôt distant pour le copier en local.
- forker : on fork un dépôt distant pour avoir une copie du dépôt distant sur notre dépôt distant. Après le fork, ce dépôt nous appartient, nous pourrons donc le modifier.

Forker un dépôt

Il n'y a pas de commande pour "forker" un dépôt, cela doit être fait dans GitHub :



Il apparaît ensuite dans nos dépôts :



Forker un dépôt

Comme il nous appartient maintenant, on peut le cloner pour le récupérer en local :

```
git clone https://github.com/RemiBoutteau/  
lighthouse.git
```


Voir le travail existant

Pour voir qui a fait quoi, on peut utiliser `git log`, mais dans des projets importants, il y a trop de commits pour s'y retrouver. Si on veut savoir qui a fait quoi, on peut trier les commits par contributeurs en utilisant :

```
git shortlog
```

On peut également faire un tri par nombre de commits grâce aux flags `-s` (nombre de commits) et `-n` (pour les trier par nombre plutôt alphabétiquement) :

```
git shortlog -s -n
```

On peut également afficher les commits par auteur avec :

```
git log --author="Paul Irish"
```

Il peut être intéressant également d'afficher les commits avec un mot particulier :

```
git log --grep="bug"
```

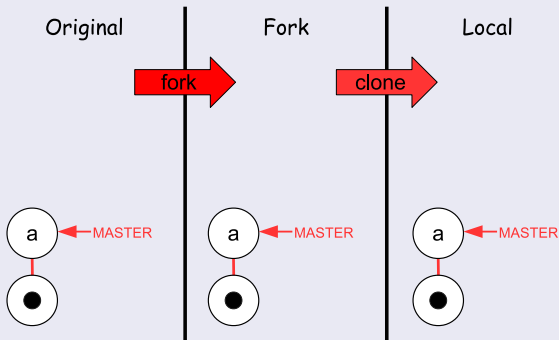
Contribuer à un projet

Pour savoir sur quoi contribuer, il y a deux choses à regarder :

- lire le fichier CONTRIBUTING.md.
- regarder les "issues" du projet.

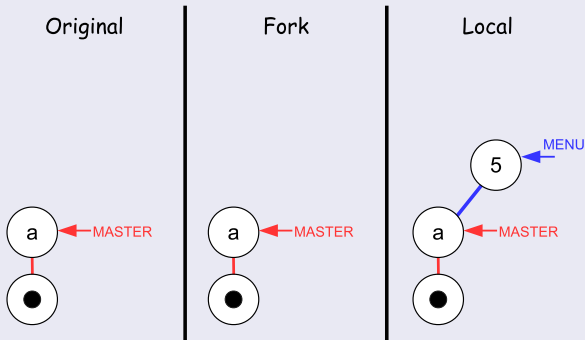
Forker un dépôt

Nous sommes maintenant dans la configuration suivante :



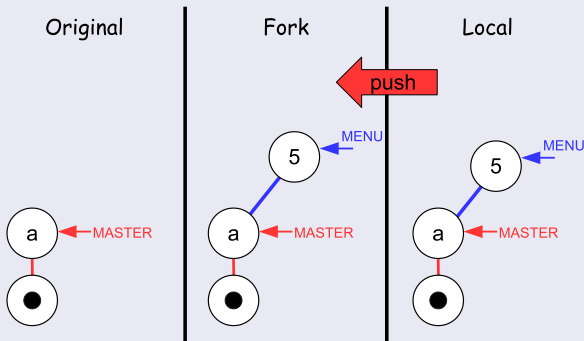
Contribuer

On a apporté des évolutions sur le projet en local :



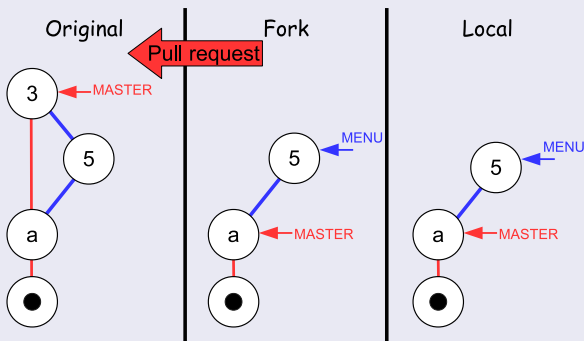
Contribuer

On fait un push pour le mettre sur notre dépôt "forké" :



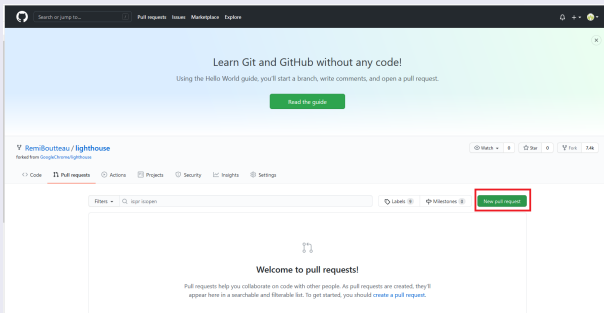
Contribuer

Puis il faut faire un "pull request". Si la requête est acceptée par le propriétaire du dépôt original, un commit de merge est créé et le pointeur de tête pointe maintenant sur ce commit :



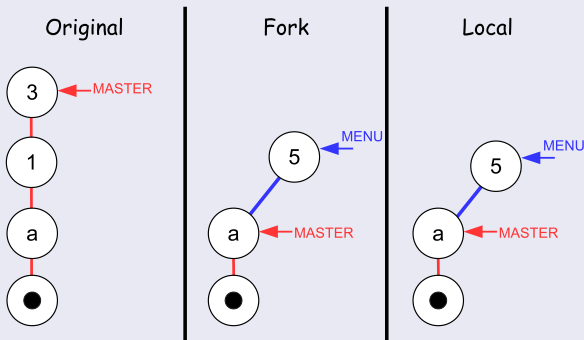
Contribuer

La demande de "pull request" se fait directement depuis l'interface de github :



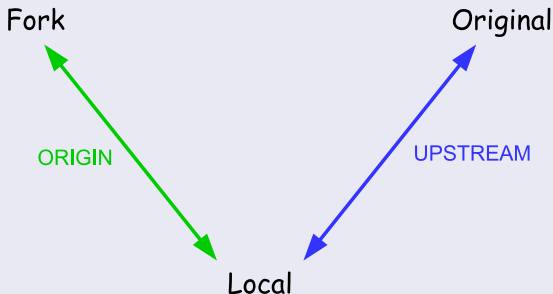
Rester synchronisé avec le dépôt original

Pendant qu'on travaillait sur le développement de notre fonctionnalité, il y a pu y avoir de nouveaux commits sur la branche principale du dépôt original. Ces commits ne sont pas mis à jour automatiquement dans le dépôt "forké".



Rester synchronisé avec le dépôt original

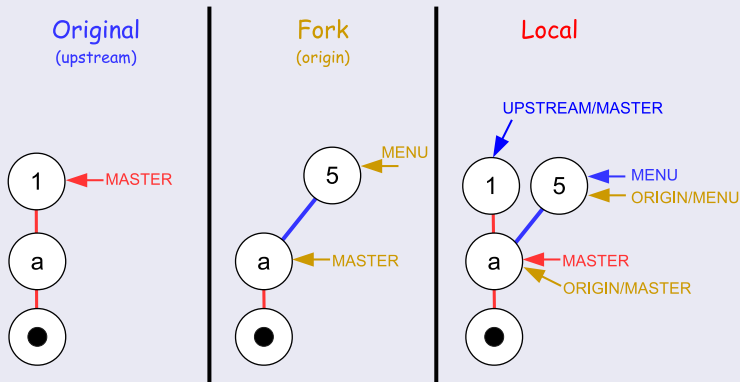
Nous avons déjà un lien entre le dépôt local et le dépôt forké (origin). Nous allons devoir ajouter un deuxième lien entre le dépôt local et le dépôt original, appelé classiquement "upstream".



```
git remote add upstream https://... # mettre l'URL  
du depot original
```

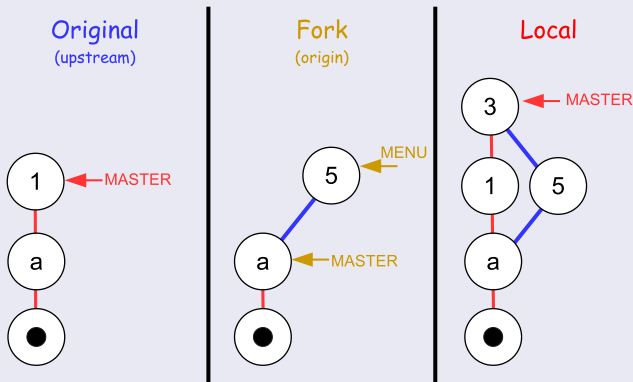
Rester synchronisé avec le dépôt original

Etat des lieux :



Rester synchronisé avec le dépôt original

Il faut donc faire un merge pour fusionner ces changements :



Il faudra également les pousser sur notre dépôt distant.

Rester synchronisé avec le dépôt original

Pour récapituler :

```
# On se met sur la bonne branche
```

```
git checkout master
```

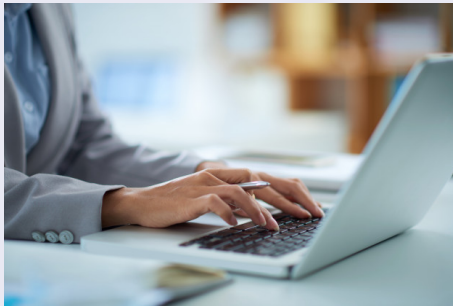
```
# On merge les changements du depot original
```

```
git merge upstream/master
```

```
# On envoie les changements sur notre depot distant
```

```
git push origin master
```

Il ne vous reste plus qu'à pratiquer.



Merci pour votre attention.

Et n'oubliez pas :

IN CASE OF FIRE 

 1. git commit

 2. git push

 3. git out!