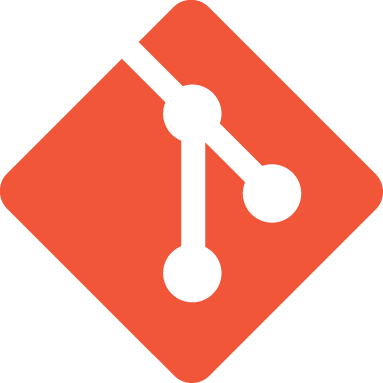
**COURS GIT**

Utilisation de

GIT

**TABLE DES MATIÈRES**

1. [Introduction 3](#_bookmark0)
2. [Fonctionnement 3](#_bookmark1)
   * 1. [Vocabulaire 4](#_bookmark3)
     2. [Fonctionnement général 4](#_bookmark4)
     3. [Détails de fonctionnement 5](#_bookmark5)
        1. [Création de projet (vers serveur) 5](#_bookmark6)
        2. [Récupération d'un projet existant (depuis serveur) 7](#_bookmark7)
   1. [Les branches et le tronc (master) 8](#_bookmark8)
      1. [Création de branches 8](#_bookmark9)
      2. [Fusion de branches 9](#_bookmark10)
      3. [Divers 9](#_bookmark11)
3. [Les bonnes pratiques 10](#_bookmark12)
   1. [Méthode GIT Flow 10](#_bookmark13)
   2. [Gitignore 11](#_bookmark15)
4. [Utilisation d'outils 12](#_bookmark16)
   1. [GitKraken 12](#_bookmark17)
   2. [Atlassian SourceTree 13](#_bookmark18)
   3. [Visual Studio et extension Git 14](#_bookmark19)
      1. [Github 14](#_bookmark20)
      2. [GitLab 15](#_bookmark21)

**1****INTRODUCTION**

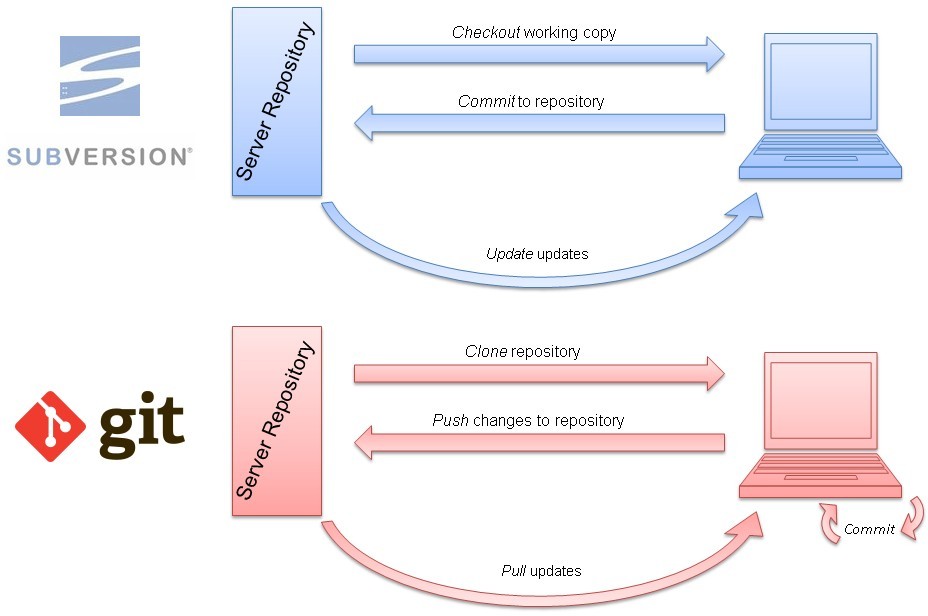
[L](https://store.unity.com/download?ref=personal)es développements nécessitent de plus en plus fréquemment de travailler en équipe, et sur différentes versions d'un même logiciel.

Il est alors difficile d'éviter les écrasements de fichiers, les modifications d'un même fichier, l'ajout d'une nouvelle fonctionnalité qui ne fonctionne pas encore, etc.

Les outils comme SVN (SubVersioN) ou GIT[1](#_bookmark2) sont les solutions les plus pratiques pour travailler de manière collaborative. Ce sont des gestionnaires de codes sources.

**2****FONCTIONNEMENT**

Bien que ce document soit orienté sur GIT, il reste intéressant de comparer avec la solution SVN.



SVN est un système centralisé dans lequel, un projet n'est hébergé que sur un seul serveur. GIT permet une décentralisation en permettant à d'autres serveurs de se synchroniser pour avoir les mêmes projets.

D'autre part, GIT permet une synchronisation locale : le développeur ne synchronise pas ses mises à jour directement sur le serveur mais dispose localement d'une gestion de version et peut choisir de synchroniser le serveur uniquement lorsqu'il aura validé plusieurs changements.

## Dans les deux cas, il est impératif d'effectuer une synchronisation avant de travailler sur les fichiers, de manière à récupérer les sources les plus récentes.

1 GIT n'est pas un acronyme, c'est Linus Torvalds qui l'a baptisé ainsi. Mais rien ne vous empêche de lire ceci : <https://en.wikipedia.org/wiki/Git>

# Vocabulaire

Ces notions sont à réutiliser au cours de cette lecture afin de mieux comprendre les termes qui sont relativmeent barbares lors de la première approche.

**Repository** : Dossier contenant le code, on distingue le "local repository" et le "distant repository".

**Fetch** : Récupération du code contenu dans le serveur distant ou une branche sans le fusionner avec le code local

**Merge** : Fusionne les branches /le code du serveur distant avec le code local.

**Pull** : Fetch puis merge

**Commit** : Groupe de fichiers englobés sous un tag précis

**Push** : Envoi des derniers commits sur le serveur distant

**Branch** : Version parallèle d’un repository permettant de travailler sur des fonctionnalités sans modifier directement le repository

**Blame** : dernière modification de chaque ligne de fichier contenant généralement auteur modification effectuée et heure, ce qui est très utile pour déterminer à qui incombe le bug et surtout à quelle modification celui-ci a été introduit.

# Fonctionnement général

# Détails de fonctionnement

Le fonctionnement dépend de la création d'un projet, ou bien de la récupération d'un projet. Après avoir ouvert un compte sur un serveur de dépôt GIT et installé une solution comme TortoiseGIT ou Kraken, il est possible de transférer les documents depuis le workspace vers le dépôt ou l'inverse.

### Création de projet (vers serveur)

Cela commence par un répertoire de travail sur le poste client. Les commandes GIT sont les suivantes :

git init git status

git add <dossier>

Les fichiers à envoyer sont maintenant indexés et l'arborescence est donc stockée deux fois (dans le workspace – *votre dossier local* – et dans l'index – *aussi appelé staging area* – )

Pour transférer les fichiers de la zone de staging vers le dépôt local (head), il faut utiliser la commande :

git commit

Cette commande créée une version de chaque fichier, comme un historique.

Enfin, l'envoi des fichiers et dossiers représentant chaque "commit" (validation ou version) est :

git push origin master

Les serveurs sont maintenant à jour (s'il n'y a pas de conflits).



**Dossier travail**

(workspace)

**Zone de transit**

(staging / index)

**Dépôt local**

(repository)

**Dépôt distant**

(repository)

**git init**

**git add**

**git commit** -m "*commentaire*"

**git commit -a**

**git push**

Il existe quelques opérations complémentaires.

Pour copier un fichier ou un dossier de l'index (staging) vers le workspace, c'est la commande :

git checkout

Cette commande écrase les modifications locales (workspace) .

Il est aussi possible de reprendre un fichier du dépôt local vers l'index avec la commande :

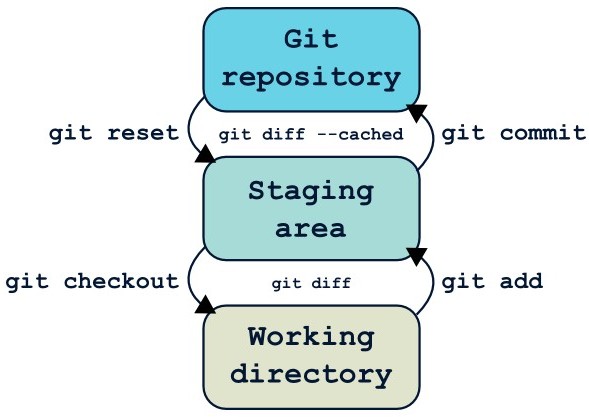
git reset

En réalité, cette commande permet d'éviter que la commande 'git commit' n'écrase un fichier finalement meilleur sur le dépôt local (zone head).

Désormais, il est possible de comparer le répertoire de travail (workspace) et la zone d'index (staging) :

git diff

Cette commande fonctionne plus efficacement sur des documents en mode texte.



En résumé : vous travaillez sur vos fichiers (Working Directory), vous poussez vos mises à jour localement et lorsque vous avez terminé, vous poussez vers le serveur distant.

image du site <https://www.miximum.fr/blog/enfin-comprendre-git/>

*Tableau 2.1 : rappel des commandes git*

### Récupération d'un projet existant (depuis serveur)

Il s'agit cette fois de partir d'un espace de travail vide et d'y placer les fichiers et dossiers tel qu'ils sont sur le serveur.

La commande qui suit permet cela :

git clone username@server:/path/to/repository.git

La commande permet de se connecter au serveur 'server' avec l'identifiant 'username' pour récupérer l'arborescence du dépôt. On peut spécifier un protocole de transfert (git clone ssh://user@host/path).

Lorsqu'il s'agit seulement de récupérer les derniers changements d'un projet existant, la commande est alors :

git pull

Cette commande essaye alors de fusionner les mises à jour du serveur avec celles que le développeur a modifié. En cas de modification des deux côtés, la commande suivante effectue un mélange automatique dans le dépôt local (et le workspace)

git pull --rebase origin master

Attention, il peut être utile d'écraser les modifications résidentes dans le dépôt local : il est donc intéressant de connaître les commandes suivantes :

git fetch origin

git reset --hard origin/master

Cela supprime tous les changements locaux en prenant le dernier historique depuis le serveur.



**Dossier travail**

(workspace)

**Zone de transit**

(staging)

**Dépôt local**

(repository)

**Dépôt distant**

(repository)

**git clone**

**git fetch**

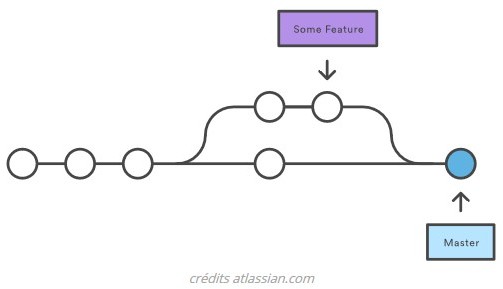
**git merge**

**git pull**

## LES BRANCHES ET LE TRONC (MASTER)

Un des intérêts majeurs de la solution GIT, est la gestion de branches.

Les branches sont utilisées pour développer des fonctionnalités isolées, comme une fonction d'impression dans un logiciel. Le logiciel fonctionne sans cette fonction et cette fonction n'est pas encore testée et validée. Le tronc peut donc recevoir des corrections mais ne contient pas la branche 'impression'.

Lorsque enfin la fonction d'impression est correcte, on peut fusionner le tronc et la branche.

# Création de branches

La création d'une branche est très simple :

git branch testBranche git checkout testBranche

La première instruction créé la branche *testBranche*.

La seconde instruction déplace l'index (head) sur la branche.

On peut aussi utiliser la commande unique git checkout -b testBranche.

Pour retrouver la branche sur laquelle est placée l'index, on utilise la commande suivante :

git branch

qui retourne

\* master testBranche

# Fusion de branches

Après avoir créé du code sur la branche, il faut intégrer ces modifications. Il faut donc **se placer** sur la branche de réception (ou sur le master), puis on va **mélanger** les deux et enfin **effacer** la branche inutile.

git checkout master git merge testBranche

git branch -d testBranche

La dernière ligne a pour effet de supprimer la branche qui ne sert plus.

Note : en cas de problème, il peut être nécessaire de forcer le commit avec la commande

git commit -a -m "*commentaire*"

# Divers

Il existe d'autres commandes pour l'usage de GIT, cependant, ils sortent du cadre du BTS.

Bien entendu, la curiosité reste le maître-mot pour ce genre de technologie : vous pouvez consulter les liens suivants :



Atlassian RogerDudler Grafikart.fr

<https://www.atlassian.com/git/tutorials/comparing-workflows> <http://rogerdudler.github.io/git-guide/index.fr.html> <https://www.grafikart.fr/formations/git/>

**3 LES BONNES PRATIQUES**

L'utilisation d'un système gérant les versions et le partage massif de fichier impose l'usage de quelques règles simples.

* 1. **MÉTHODE GIT FLOW**

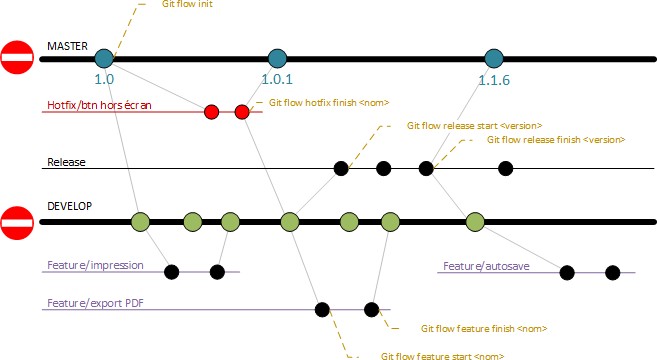
Le "GIT flow[2](#_bookmark14)" est une recommandation pour faciliter le développement en conservant la qualité et les fonctionnalités. Il n'y a rien de pire que de devoir retourner en arrière à cause de régressions.

Il y a donc deux lignes principales : MASTER et DEVELOP. Les développeurs ne sont pas autorisés à coder directement dessus.

La ligne MASTER est la ligne publique, la version que les utilisateurs peuvent employer. Si un bug est détecté, des développeurs vont créer un "hotfix" concernant le bug.

La ligne DEVELOP permet aux développeurs ou aux personnes qui travaillent sur les documents et codes de démarrer des améliorations (features) : cela évite les conflits de trop nombreux codeurs sur les mêmes fichiers sources.

La branche release est particulière, car elle assure le lien entre la ligne DEVELOP et la ligne MASTER : c'est cette branche que les utilisateurs avertis pourront tester, sous la forme d'une version beta.



La plupart de ces commandes sont difficiles à mémoriser, mais il existe une surcouche qui facilite ces manipulations (commandes en jaunes sur le schéma).

2 <https://blog.nathanaelcherrier.com/2016/07/11/gitflow-la-methodologie-et-la-pratique/>

## GITIGNORE

Gitignore est un fichier de configuration lu par GIT. En général, on y trouve les fichiers qui seront ignorés lors des opérations de synchronisation.

Par exemple, les dossiers *bin* et *obj* générés par Visual Studio et C# ne doivent pas êtres synchronisés (ils contiennent le résultat des compilations, c'est-à-dire les fichiers exécutables).

Vous pouvez également ignorer les fichiers commençant par ~ ou finissant par .tmp.

Voici un exemple de fichier .gitignore pour Visual Studio (notez la présence d'expressions régulières simples) :

## Ignore Visual Studio temporary files, build results, and ## files generated by popular Visual Studio add-ons.

##

## Get latest from https://github.com/github/gitignore/blob/master/VisualStudio.gitignore

# User-specific files

\*.suo

\*.user

\*.userosscache

\*.sln.docstates

# User-specific files (MonoDevelop/Xamarin Studio)

\*.userprefs

# Build results [Dd]ebug/ [Dd]ebugPublic/ [Rr]elease/ [Rr]eleases/ x64/

x86/ bld/ [Bb]in/ [Oo]bj/ [Ll]og/

Il s'agit d'un extrait mais vous trouverez l'original et d'autres modèles de fichiers sur <https://github.com/github/gitignore>

La syntaxe reste simple :

* \*.log ► supprime tous les fichiers dont l'extension est log (quel que soit leur emplacement)
* \*\*/logs ► ne synchronisera pas les fichiers dans les répertoires logs
* # ► indique un commentaire

**4 UTILISATION D'OUTILS**

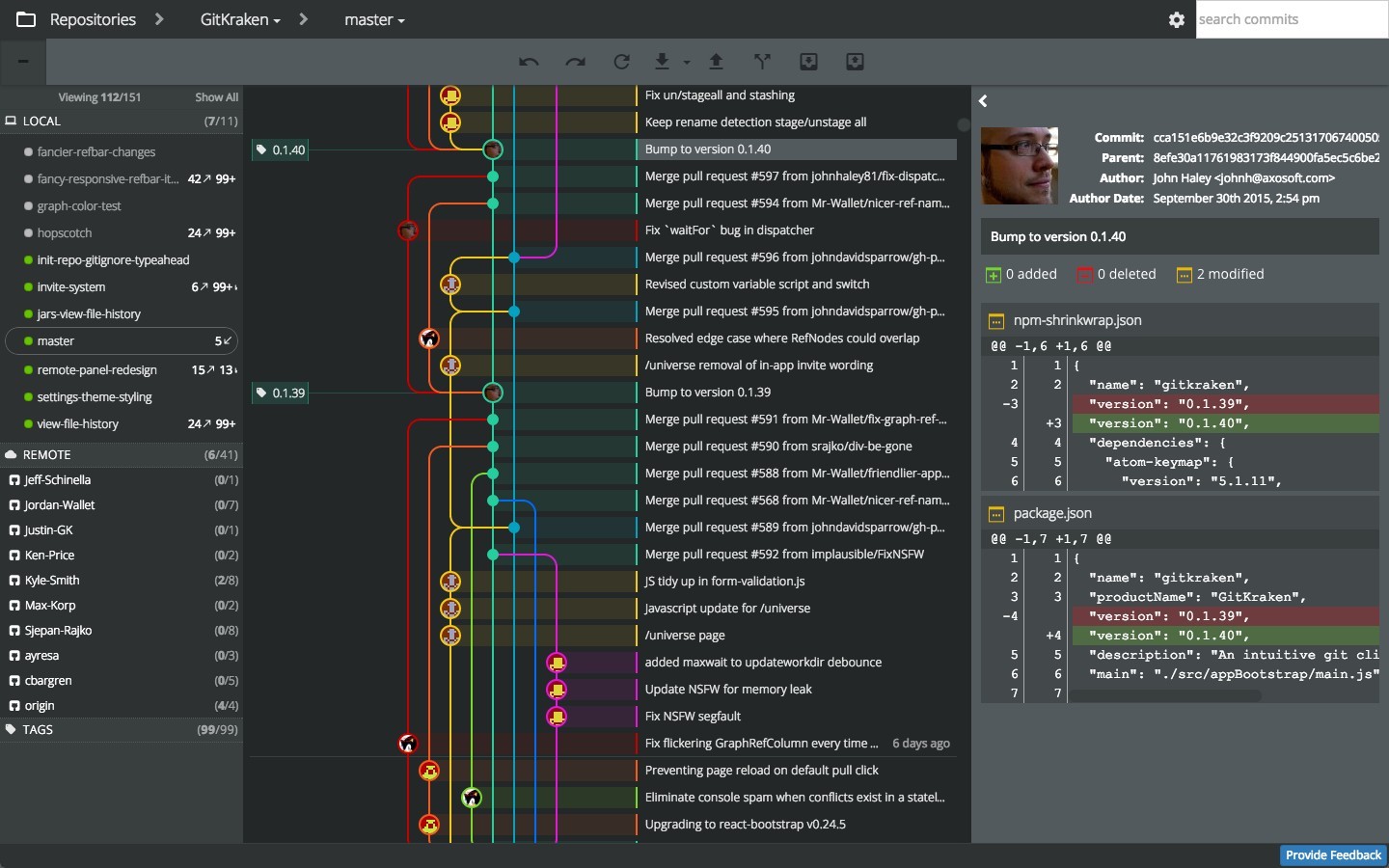
La gestion en ligne de commande est intéressante mais c'est un peu comme développer sous vi ou vim : tôt ou tard, la complexité implique l'usage d'éditeurs plus élaborés.

Cela ne signifie pas l'oubli des commandes, car les systèmes graphiques peuvent parfois ne plus être capables de gérer les conflits ou problèmes et il faudra alors remettre le système opérationnel… manuellement !

## GITKRAKEN

GitKraken est basé sur le framework Electron : son interface est donc sobre, agréable. Il fonctionne sur plusieurs OS comme Microsoft, Mac et Linux. Il n'est pas open-source.

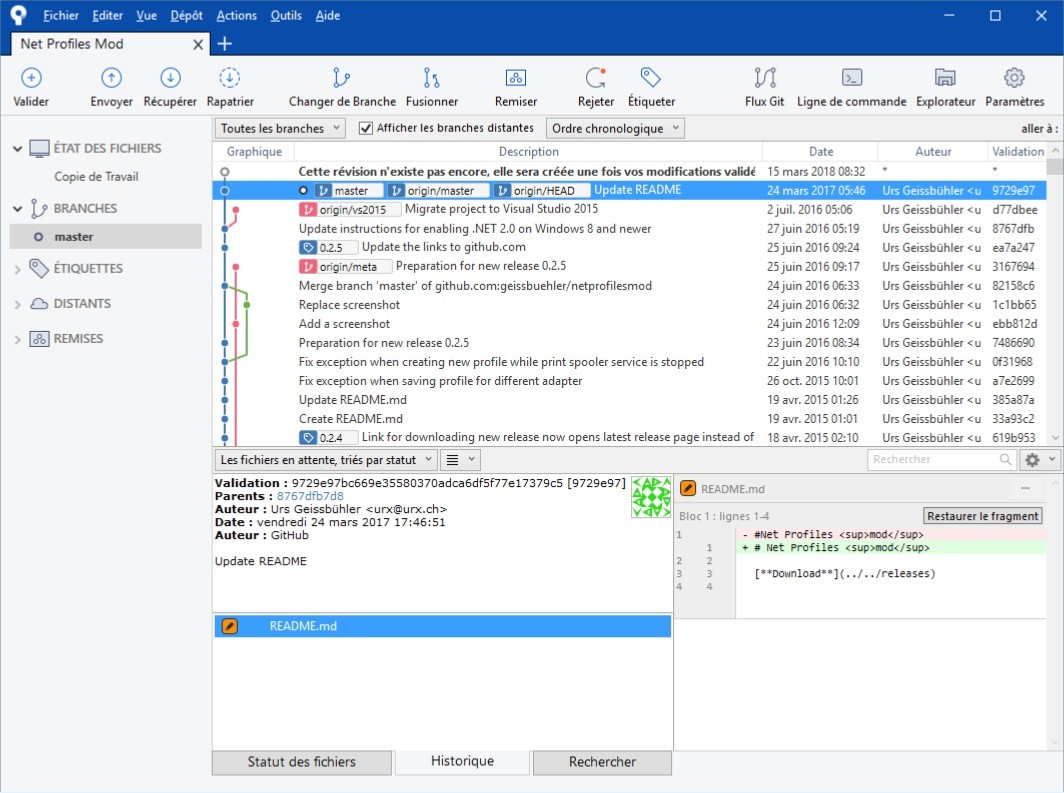
Il nécessite un enregistrement préalable sur les serveurs de KitKraken pour pouvoir utiliser la version gratuite. J'ai rencontré quelques soucis avec le proxy du lycée.



## ATLASSIAN SOURCETREE

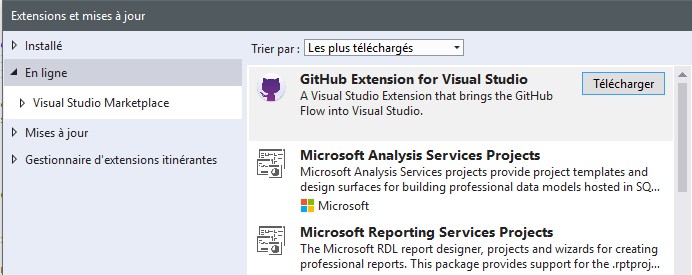
Cet éditeur est relativement simple et est gratuit également.

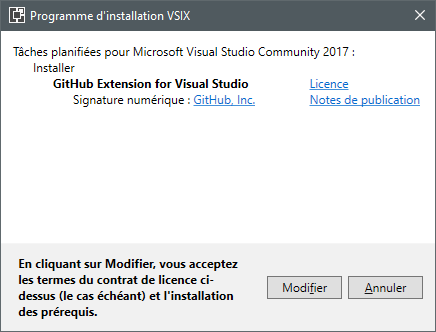
Son fonctionnement est similaire à GitKraken et il nécessite également enregistrement préalable sur les serveurs de Atlassian pour pouvoir fonctionner.



## VISUAL STUDIO ET EXTENSION GIT

# Github

L'environnement VS2017 community n'intègre pas git par défaut, mais il existe une extension :

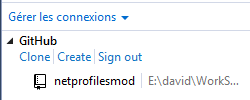
Une fois téléchargée et installée (nécessite la fermeture de VS), vous devez utiliser vos identifiants github.

Allez dans le menu "Équipe" (si VS2017 en anglais, alors "Team".

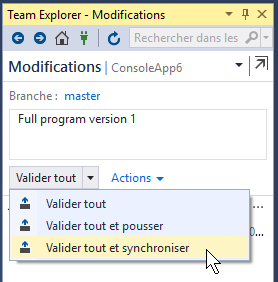


Dans le panneau de droite, cliquez sur "Connect…" et saisissez vos identifiants.

Désormais, vous pouvez utilisez GitHub pour vos projets.

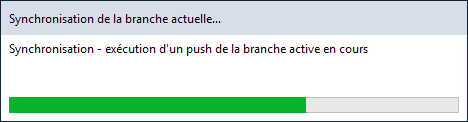
Cliquez sur "Create" dans l'explorateur à droite, puis choisissez le nom de votre projet, le répertoire local et une licence. Voici un lien pour vous aider : <https://choosealicense.com/>

Lorsqu'une modification paraît correcte et prête à être envoyée sur le serveur GitHub, validez et synchronisez tout (placez obligatoirement un commentaire) :

N'oubliez pas de revenir au menu accueil de l'outil en cliquant sur l'icône maison .

La différence entre Valider tout et Valider tout et synchroniser est la suivante :

* Valider Tout synchronise localement
* Valider tout et synchroniser envoie vers github (push) puis effectue la synchronisation des changements en retour.

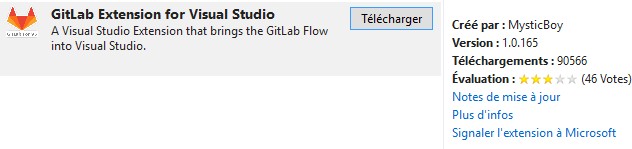


[https://blogs.msdn.microsoft.com/benjaminperkins/2017/04/04/setting-up-and-using-github-in-](https://blogs.msdn.microsoft.com/benjaminperkins/2017/04/04/setting-up-and-using-github-in-visual-studio-2017/) [visual-studio-2017/](https://blogs.msdn.microsoft.com/benjaminperkins/2017/04/04/setting-up-and-using-github-in-visual-studio-2017/)

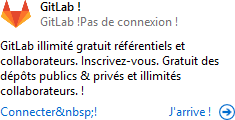
<https://www.youtube.com/watch?v=Nw2tfO44_9E>

# GitLab

Comme pour Github, il faut ajouter une extension appelée "GitLab extension for Visual Studio".

Téléchargez l'extension et installez-la (6,6 Mo environ).

Fermez Visual Studio, l'installation peut alors se faire.

Une fois dans le bandeau à droite, sélectionnez "Connecter !" (même s'il y a un bug HTML).

De mon côté j'ai rencontré *un problème lié aux dépendances NewtonSoft.Json*…

[https://marketplace.visualstudio.com/items?](https://marketplace.visualstudio.com/items?itemName=MysticBoy.GitLabExtensionforVisualStudio) [itemName=MysticBoy.GitLabExtensionforVisualStudio](https://marketplace.visualstudio.com/items?itemName=MysticBoy.GitLabExtensionforVisualStudio)