GIT FOR DUMMIES (and LD)

SOMMAIRE

Table des matières

Historique

Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. En 2016, il s'agit du logiciel de gestion de versions le plus populaire qui est utilisé par plus de douze millions de personnes.

Similaire en cela à BitKeeper, *Git* ne repose pas sur un serveur centralisé. C'est un outil de bas niveau[réf. souhaitée], qui se veut simple et performant, dont la principale tâche est de gérer l'évolution du contenu d'une arborescence.

Git indexe les fichiers d'après leur somme de contrôle calculée avec la fonction de hachage SHA-1. Quand un fichier n'est pas modifié, la somme de contrôle ne change pas et le fichier n'est stocké qu'une seule fois. En revanche, si le fichier est modifié, les deux versions sont stockées sur le disque.

Contrastant avec les architectures de logiciel de gestion de versions habituellement utilisées jusqu'alors, *Git* repose entièrement sur un petit nombre de structures de données élémentaires. Linus Torvalds expliquait ainsi : « Par bien des aspects, vous pouvez considérer *git* comme un simple système de fichiers. Il est adressé par contenu, et possède la notion de versionnage, mais je l'ai vraiment conçu en prenant le point de vue d'un spécialiste des systèmes de fichiers (après tout, j'ai l'habitude de travailler sur des noyaux) et je n'avais absolument aucune envie de créer un système de gestion de version traditionnel. »5 Les premières versions de *Git* offraient une interface rudimentaire pour manipuler ces objets internes avant que les fonctionnalités courantes de gestion de version ne soient ensuite progressivement ajoutées et raffinées.

Git est considéré comme performant, au point que certains autres logiciels de gestion de version (Darcs, Arch), qui n'utilisent pas de base de données, se sont montrés intéressés par le système de stockage des fichiers de *Git* pour leur propre fonctionnement6,7. Ils continuent toutefois à proposer des fonctionnalités plus évoluées.

Dès le début, *Git* a été pensé dans le but de fonctionner de façon décentralisée, c'est d'ailleurs l'une des clefs de son succès[réf. souhaitée]. La décentralisation de *Git* a aussi beaucoup apporté au développement des logiciels libres, puisque le besoin de demander un compte sur un dépôt SVN ou CVS centralisé devient obsolète. Il suffit de forker un projet ou de le cloner pour commencer à travailler dessus (avec tout l'historique du projet en local) et ensuite de proposer sa contribution (pull request) au dépôt principal (mainteneur principal du projet).

Les serveurs *Git* utilisent par défaut le port 9418 pour le protocole spécifique à *Git*. Les protocoles HTTP, HTTPS et SSH (et leurs ports standards) peuvent aussi être utilisés.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Non mais sans déconner t'as vraiment cru que j'allai faire un cours d'histoire sur git ?
Voilà... Le ton est pausé. Passons au véritable cours.

TLDR, Pourquoi *Git* ?

Parceque les clefs USB c'est chiant à envoyer par la poste.

Parceque Google Drive ne permet pas de modifier un fichier chacun de notre côté ni de "fusioner" les modifications intelligement. Aucune possibilité de revenir en arrière. Et parceque tout ça dépend de Big Brother.

SVN ? Alors oui mais non. SVN est en effet assez similaire à *git* en bien des points, mais moins flexible car moins de possibilité au niveau des branches, pas de gestion de commit en local git push, git pop (oui il est possible de faire des patchs mais c'est horrible sur le long terme), un push <=> une copie INTÉGRALE du projet à chaque fois.... Ca prend de la place .... c'est chiant de trouver un host ... Et c'est soooo 2000.

Là où *git* lui va partir d'un projet de base (vierge) et à chaque push il va faire la DIFFÉRENCE par rapport à la version précédente. C'est un gain de temps (en upload/download), un gain d'espace et beaucoup plus facile à manipuler.

ATTENTION : *Git* fonctionne très bien pour du code, du texte. Mais gère très mal les images, sons, scènes, tout ce qui est binaire, illisible en ouvrant un fichier texte.

« Oui mais alors les scènes sous unity c'est pas du binaire, ca devrait passer en théorie ». Alors oui c'est possible de merger les scènes à la main… Mais le premier qui évoque cette possibilité perdra sa dite main. Car c'est tout simplement illisible impossible de dire que la maison que Marc a placé hier au coin de la rue Colombin correspond au prefab id : « 2548FGnbr » placé aux coordonnées: « 35145/54321/3521 » avec les attributs « erazhetrfdghtrdcvgtrfd ». Et étant donné que la moindre modification risque de changer l'intégralité du fichier à chaque moment …. Non juste non, travaillez un seul à la fois par scène…

Je m'excuse auprès de quiconque lis ce pdf en pensant y trouver un cours précis sur Git. Ce document a été rédigé dans le seul but d'aider les Level Designers avec qui je travaillais sur mon projet scolaire. Ce cours n'est donc pas exhaustif, rentre au minimum dans les détails techniques et use de raccourcis à s'en arracher les cheveux pour ceux qui connaissent réellement git. Ce document est à considéré comme un support de cours afin d'initier les gens / ld à la magie de git.

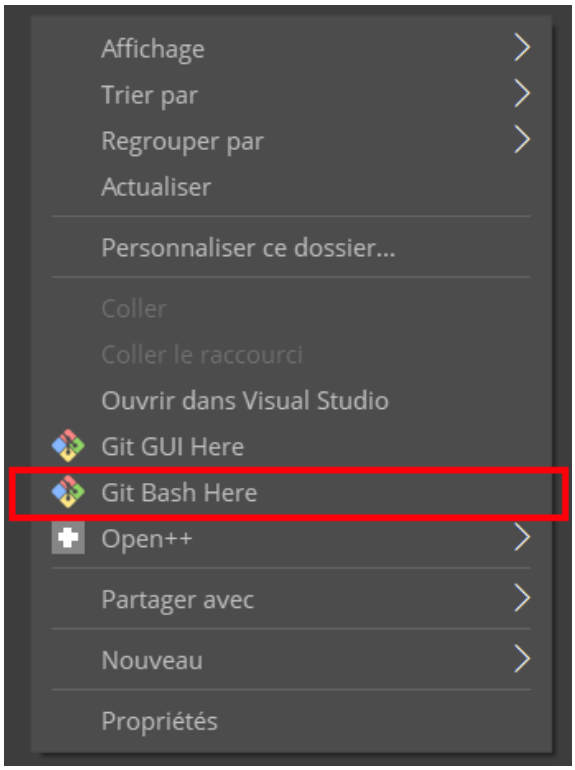
Git-Scm Bash

Où le télécharger ? https://git-scm.com/

« Pk bash ? Jém pa ya pl1 2 mo compliké » - Robert Edouard, Gamagora 2017

Libre à toi Robert de partir sur un outil plus visuel, les idées devraient être les mêmes. Mais n'attend pas d'aide de ma part. De plus tu trouveras le *git* bash PARTOUT où *git* est installé et quel outil visuel est plus spécifique à un environnement, un projet, prompt à disparaître, prompt à appliquer ses automatismes assez obscurs, etc

Une fois installé, il vous suffit de faire « clic droit » → « Git Bash Here » dans n'importe quel dossier pour ouvrir la console Git-Bash.

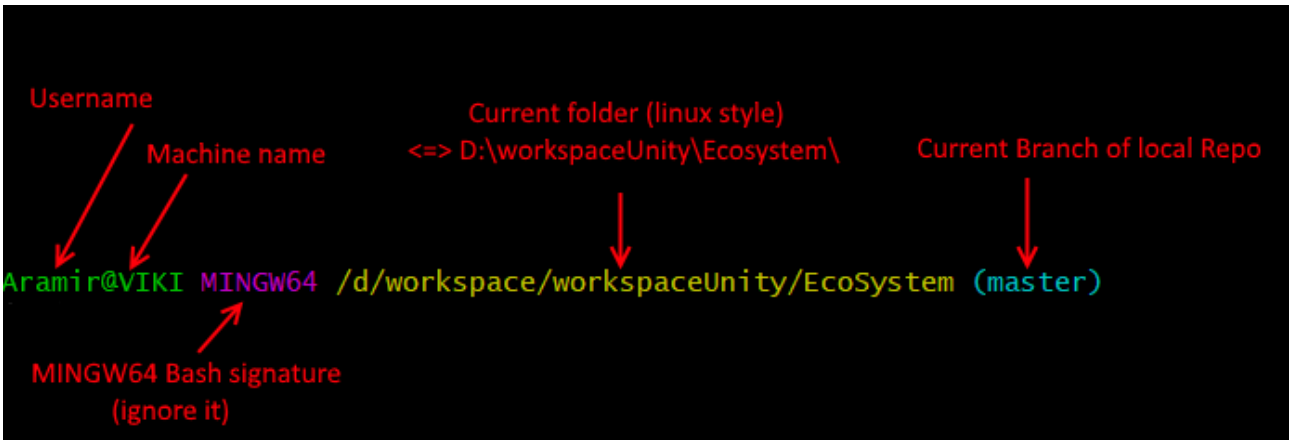


Menu contextuel git-scm

Normalement vous devriez vous trouvez face à votre pire cauchemar, une console textuelle. Respirez un bon coup, tout va bien se passer.

Voici quelque commande pour s'en sortir :

- **help** : va lister l'intégralité des commandes disponibles
- **ls** : va lister les fichiers et dossier se trouvant dans le répertoire courant (adresse similaire à windows à ceci prêt que les '\' deviennent des '/' et que la lettre du disque dur n'est pas suivi par ' :'
- **cd** : permet de rentrer dans un dossier
- **cd ..** : permet de revenir au dossier précédent
- Il est possible d'utiliser « **flèche haut** », « **flèche bas** » pour naviguer dans l'historique des commandes tapées précédemment… Pratique pour éviter d'avoir à retaper ce mega commentaire de 5 lignes après une erreur.
- Copier coller ? Git-bash est une console type « linux » par conséquent les raccourcis sont différents. Pour **copier** du texte il suffit de le surligner et pour **coller** il faut faire un clic molette.



Explication du bash Git

Avant de faire quoi que ce soit il va falloir indiquer qui vous êtes, afin que git signer votre travail, vos modifications à votre « username » et « adresse mail ». Voici les commandes pour donnez à git ces informations :

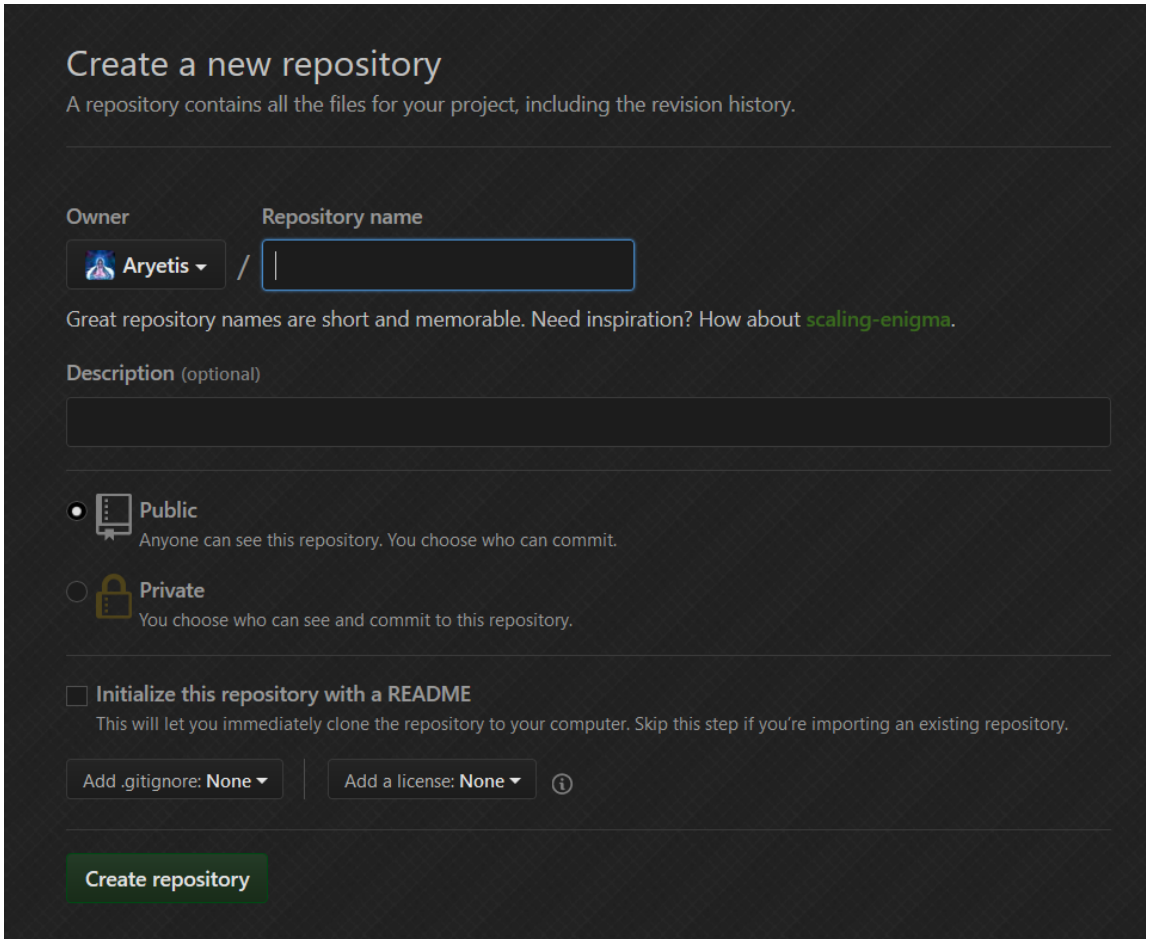
- **git config --global user.name "Mona Lisa"** : Permet d'indiquer votre username (duh)
- **git config --global user.email "*email@example.com*"** : Pour le mail

Gardez en tête que cela permet à git de « signer » votre travail, il n'est pas obligatoire de faire correspondre ces identifiants avec vos identifiants github. Et oui github, n'est qu'une plateforme, un serveur git gratuit. Le logiciel git et les serveurs github ne sont pas dépendants, il est possible d'utiliser d'autres plateformes pour stocker du code avec git (ex : gitlab, bitbucket, etc), comme il est possible d'envoyer des colis par plusieurs sociétés (ex :la poste, UPS, fedex, etc).

Github de son côté devrait vous demander vos identifiants github lorsque vous tenterez d'envoyer des données sur ses serveurs git, autrement dit lorsque vous ferez des push / pull

Créer son Repo Github

C’est bon? Ok maintenant on va créer notre propre Dépôt sur http://github.com/new

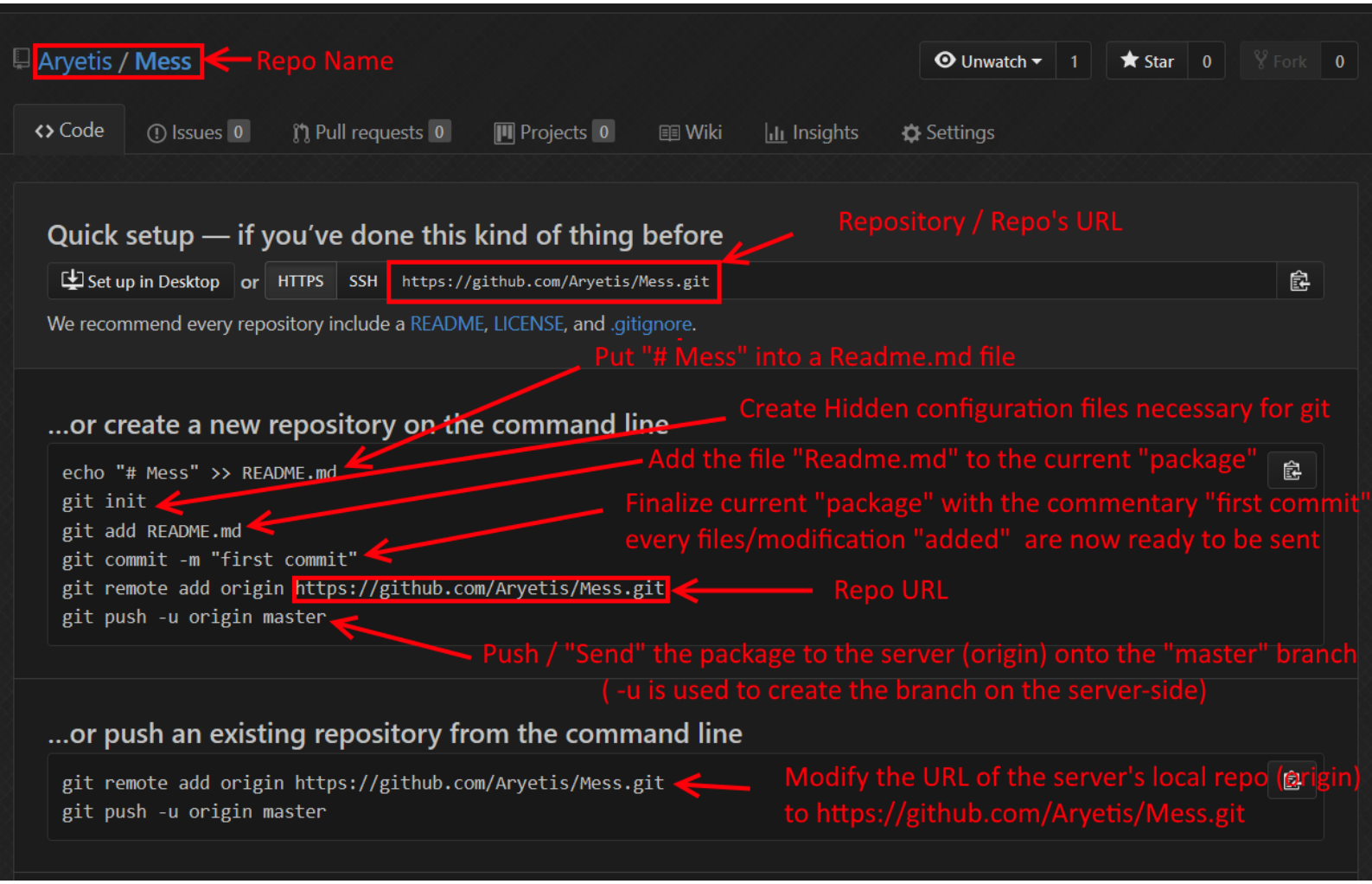


Page de crétion d'un Repo git github

Donnez lui un petit nom, une description même si vous le sentez et passons à la suite.

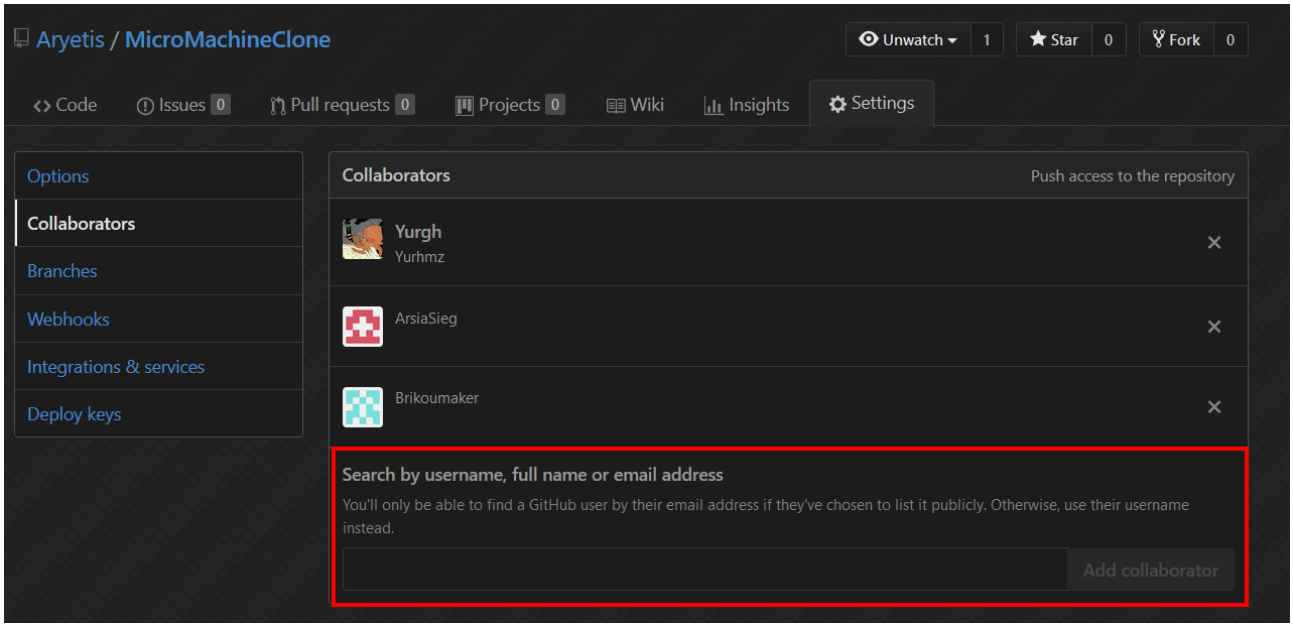
« Wallaaaaah y a plein de texte »

Alors d'une tu arrêtes de m'interrompre inconnue voix intradiégétique, on est pas sur le site du zéro ici ! Et de deux oui je sais mais je vais tout traduire pour toi.



Comment afficher les fichiers cachées ? http://bfy.tw/GAxs

L'étape suivante et de donner les droits d'accès à vos « Collaborateurs », il vous faudra vous rendre dans l'onglet Settings → Collaborators :



Github collaborators

Une fois sur cette page il vous faut ajouter vos collaborateurs en utilisant leur mail ou leur nom de profil. Attention cependant l'effet n'est pas immédiat. Il faudra que chaque collaborateur vérifie ses mails pour accepter l'invitation envoyée par github.

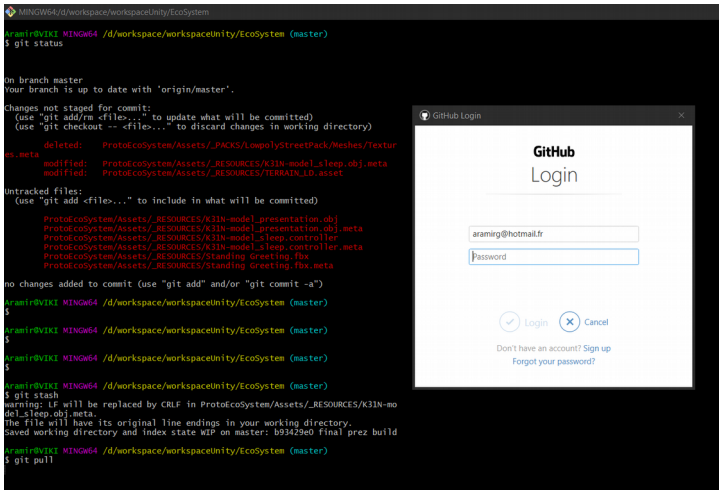
Voilà maintenant que vous avez votre propre repo *git*, côté serveur, il va falloir le remplir depuis votre machine / votre client.

Pour cela une personne va devoir :

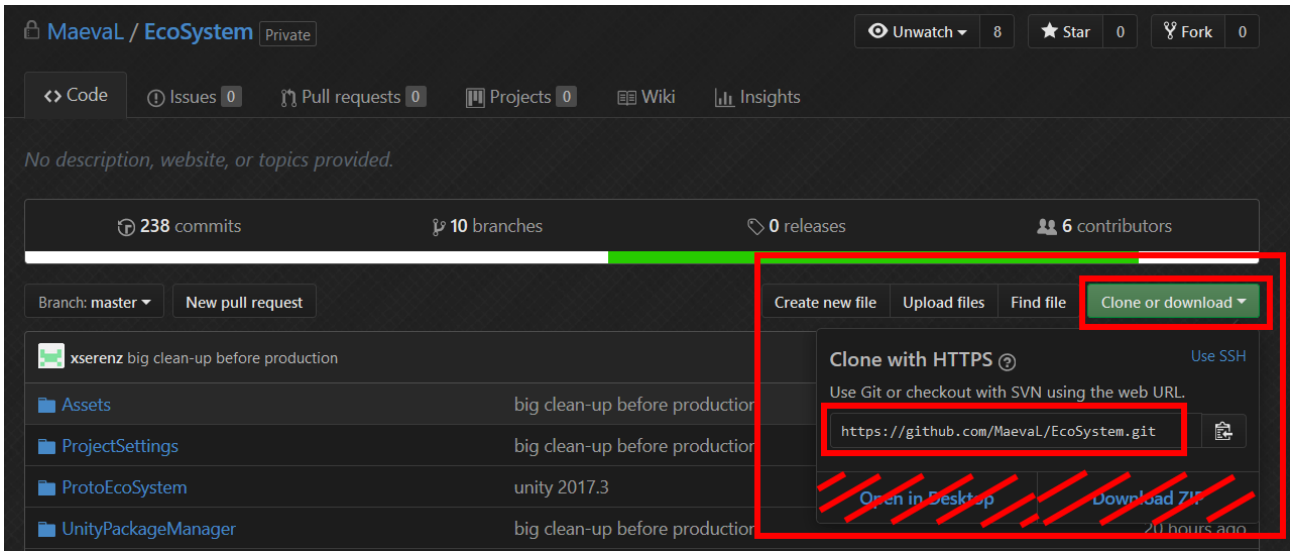
1. Créer un projet Unity vierge
2. Placer à la racine de ce projet un fichier « .gitignore » : https://pastebin.com/4zbc2nLX
   (Plus à ce sujet dans : « Stash ? Pop ? .gitignore and other Tricks » p .19 et « Annexe, Troubleshootings » p.20) Si vous ne voyez pas ce fichier, pensez à activer sous windows l'affichage des fichiers cachés et des extensions de fichiers.
3. Suivre les instructions donnez sur la page github « Quick Setup » vue précédemment.
   Il va juste falloir les adapter très légèrement pour accueillir notre projet Unity. (Promis à la fin du cours vous devriez pouvoir comprendre ces lignes sans problèmes, pour l'instant il faut me faire confiance).
   1. **Se rendre à la racine du projet Unity et faire clic droit → « Git Bash Here »**
   2. **echo « Empty Readme.md » >> Readme.md** : Va créer un fichier Readme.md. Ce fichier sert de page d'accueil internet pour votre projet github et est formaté au format « Markdown ». Renseignez vous donc sur comment vous en servir et essayez de le garder propre si vous souhaitez partager votre projet. Indiquez l'état du projet, un synopsis, des liens de téléchargement, une license, etc
   3. **git init** : va créer un Repo git local ainsi que tout les fichiers de configurations cachés
   4. **git add .** : oui je sais, sur la page github c'est marqué « git add Readme.md », mais nous, nous avons tout un projet Unity à envoyer sur le serveur et non juste un fichier Readme.md. N'oubliez pas le '**.**' !
   5. **git commit -m « first commit »** : Va empaqueter tout les fichiers, afin de les envoyer sur le serveur
   6. **git remote add origin [GITHUB_URL_REPO]** : La ligne la plus importante. À copier coller depuis la page github étant donné que je ne peux pas deviner l'URL de vos futurs Répos github (duh). Celle ligne va lier votre Repo git local à votre Repo git serveur.
   7. **git push -u origin master** : Va envoyer tout les fichiers (le projet Unity) précédemment empaqueter par la commande « git commit »

Récupérer un projet depuis Github

Très bien votre git local est au point, le git serveur est rempli. Il ne reste plus qu'à vos camarades de récupérer ce qui se situe sur le serveur et tout le monde sera prêt à travailler. Mais comment faire ? Faut-t-il que chaque personne fasse son propre serveur git, et tout fusionnera comme par magie plus tard ? NON ! Tout est disponible sur le serveur git, vous avez déjà tout mis en place. Aussi il ne reste plus à vos camarades qu'à récupérer cette « base commune ». N'UTILISEZ PAS le gros bouton vert présent sur la page github pour télécharger une archive zip du projet car cela ne va pas initialiser votre Repo git local.



Github demandant ses logins lors d'un pull



Récuperation de l'URL du Repo git serveur

À la place :

1. Envoyez l'url github de votre repo à vos collaborateurs et dites leurs de récupérer l'url du repo (pas celle de la page github!!!) En cliquant sur le bouton « Clone or download »
2. Dites leurs de créer un dossier dans lequel ils souhaitent récupérer le projet Unity puis de faire « Clic Droit » → « git bash here » et de rentrer la commande suivante
3. **git clone [REPO_URL]** : cette commande va automatiquement initialiser le Repo Git en local, créer les fichiers de configurations et récupérer ce qui se trouve sur (la branche master) du Repot git serveur

Et voilà, ça y est, le serveur est prêt, tout le monde est prêt, le travail peut commencer. Ces manipulations peuvent sembler longues la première fois mais une fois l'habitude prise le tout ne met pas plus de 10/15 minutes à mettre en place.

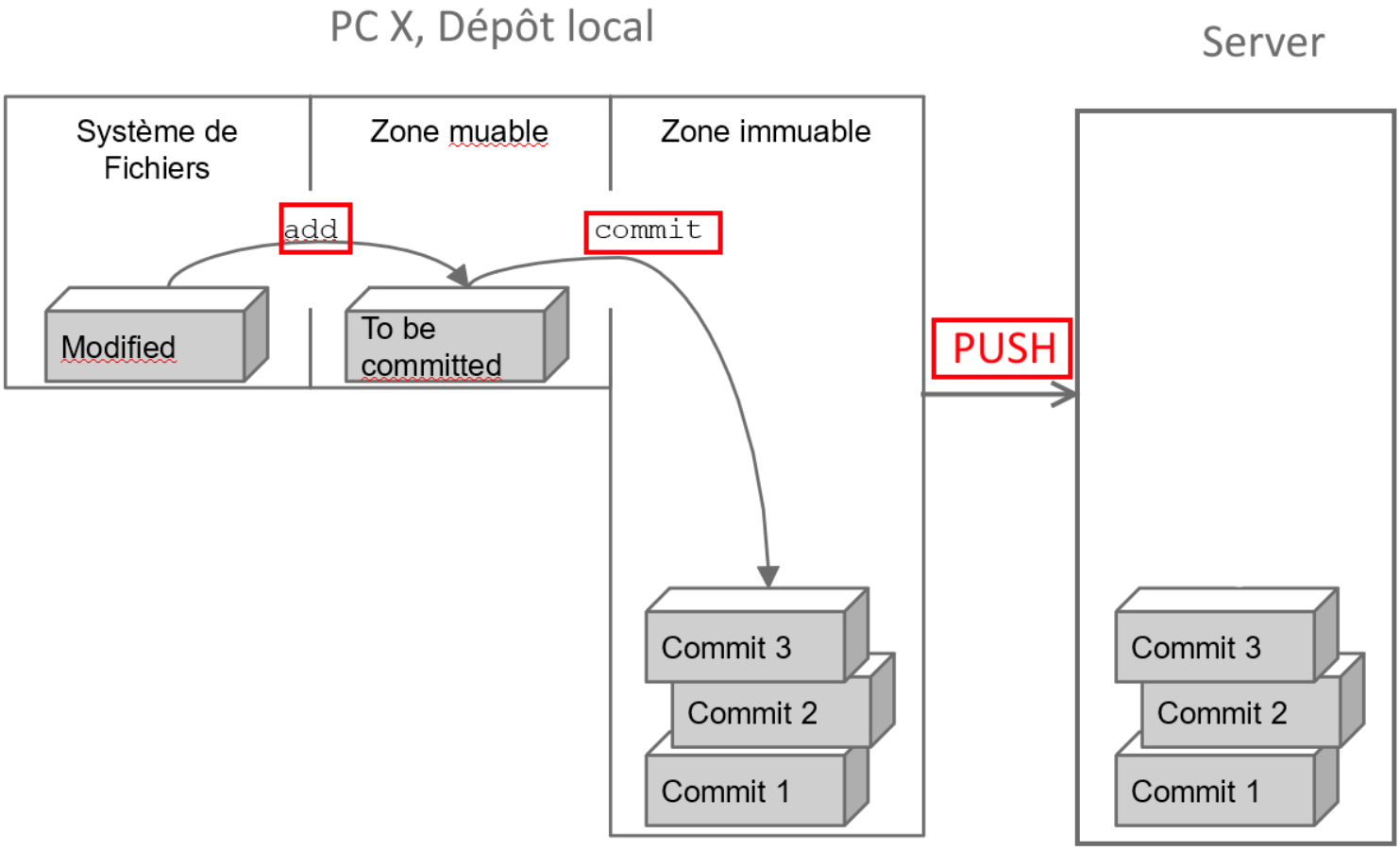
Dépôt Distant / Local

LA notion importante à comprendre dans *Git* est la notion de **Dépôt** (aussi appelé Repository / Repo).

Lorsque l'on travaille sur *git*, chaque personne travaille sur sa machine / sur **son dépôt local**, et dépose ses fichiers sur un server / le **dépôt distant**. Aussi sauf à ce que vous fassiez des manipulations avancées (ou copiez collez du code chelou récupéré sur google) vous ne pouvez pas supprimer le travail des autres. Comme *git* fonctionne en stockant les **différences** entre chaque commit / nouvelle version si une mauvaise manipulation est faîte il suffit de revenir en arrière, annuler ce commit, ce qui revient à enlever la dite différence. Ce n'est pas une raison pour foutre le souc non plus. On vous jugera aussi sur la qualité du code que vous envoyez sur github. Github c'est public les recruteurs aiment bien regarder le compte github de chaque Dev, alors pourquoi pas les LD aussi ?

Le principe derrière tout ça étant que tout le monde travaille sur une base commune fonctionnelle stockée sur le serveur. Donc le premier qui bloque les autres en envoyant sur le serveur du code qui ne compile se fait couper sa couille gauche… Bref, chacun a une base saine et travaille de son côté. Une fois le travail terminé les modifications faites en local sont appliquées sur le serveur afin que les autres puissent récupérer les changements.

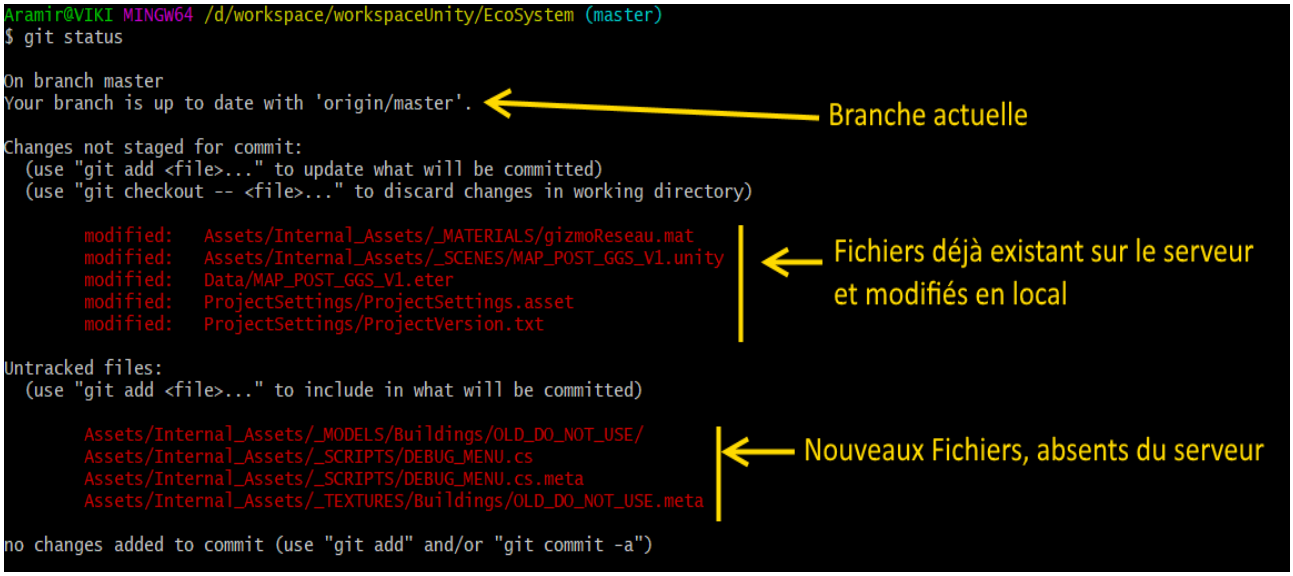
Tant que votre travail n'est pas push, il n'est pas sur le serveur ! Il n'est pas récupérable par les autres ! Vous ne pouvez PAS non plus écraser le travail de votre voisin par erreur, même pour la blague, même juste pour voir, même pour rien du tout ! Il est (presque) impossible de perdre des données une fois qu'elles ont été commit sur le serveur, il est (presque) toujours possible de revenir en arrière pour les récupérer !



Envoyer ses modifications

Pour uploader ses modifications sur le server , Jean-Kevin va effectuer la manipulation suivante :

1. **git fetch** : Cela va permettre à « git local » de consulter « git server » afin qu'il obtienne la liste des modifications effectuer sur le server, qui a pushé quoi, quelle branche a été créée, supprimée, etc. En théorie il est possible de s'affranchir de cette étape si chacun respecte les règles du jeu et ne touche pas à la branche de son voisin.
2. **git status** : Cela va afficher la liste des fichiers modifiés localement depuis le dernier push. C'est là aussi une simple étape de vérification, afin de s'assurer que l'on envoie pas n'importe quoi au serveur. On va donc consulter précisément la liste des fichiers modifiés.
3. **git add [FileName]** : (ex : git add Assets/Scenes/Game.unity) va permettre de dire à *git* « J'ai changé tel fichier et je voudrai que tu le mettes le dans le prochain paquet cadeau en partance pour le serveur ». Pour gagner du temps, JK peut aussi taper : « git add . » ce qui va automatiquement add tout les fichiers modifiés (attention de ne pas envoyer n'importe quoi)
4. **git commit -m « awesome comment explaining to my mates what I'm sending»** : Une fois tout les fichiers « ajoutés au paquêt », il faut finaliser / refermer celui-ci, mettre un joli petit nœud dessus et écrire une petite note pour le destinataire expliquant ce qu'il contient. C'est exactement ce que fait « git commit ».
5. **git push** :Une fois le paquet terminé JK, peut décider de l'envoyer sur le server ou de le garder pour lui et d'envoyer plusieurs paquets d'un seul coup plus tard. En revanche il n'est pas (facilement) possible de modifier / rajouter des fichiers à un paquet une fois celui-ci envoyé sur le serveur. Si JK veut donc rajouter d'autres fichiers ou réparer ses bêtises, il lui faudra refaire la manipulation depuis le début pour créer un nouveau « paquet » et l'envoyer. Attention cependant par défaut JK enverra ses paquets au serveur sur la branche qu'il utilise actuellement ! (cf : branche actuelle indiquée en bleu dans le git-bash). JK n'a pas a touché à la branche des gens codant l'IA si il bosse sur la branche « AntiCheat ». (Plus à venir sur les branches un peu plus tard).



Sortie habituelle donnée par git status après avoir ajouter ces fichiers à l'aide de "git add"

TLDR :

1. git fetch (facultatif mais faites le quand même au cas où)
2. git status (facultatif mais non vraiment sans déconner ça aide à s'y retrouver)
3. git add FileName (ou « git add . » si tout les fichiers modifiés sont concernés)
4. git commit -m « awesome comment explaining to my mates what I just did »
5. git push [-u BranchName]

En théorie tout ça devrait fonctionner constamment …. À condition de travailler seul…

Et oui parce que pendant que vous travaillez les autres vont eux aussi de leur côté « pusher » des choses sur le serveur. Et donc au final pour que *git* puisse faire la « différence » entre votre code et le code sur le serveur, il va devoir en premier lieu récupérer en local les modifications faites sur le serveur. « Mais dis moi Jamy, Comment on fait pour recevoir le travail des autres ? »

```
$ git push origin master
To git@github.com:Charles0429/update_test.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:Charles0429/update_test.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Erreur typique du push sans pull au préalable

Recevoir des modifications

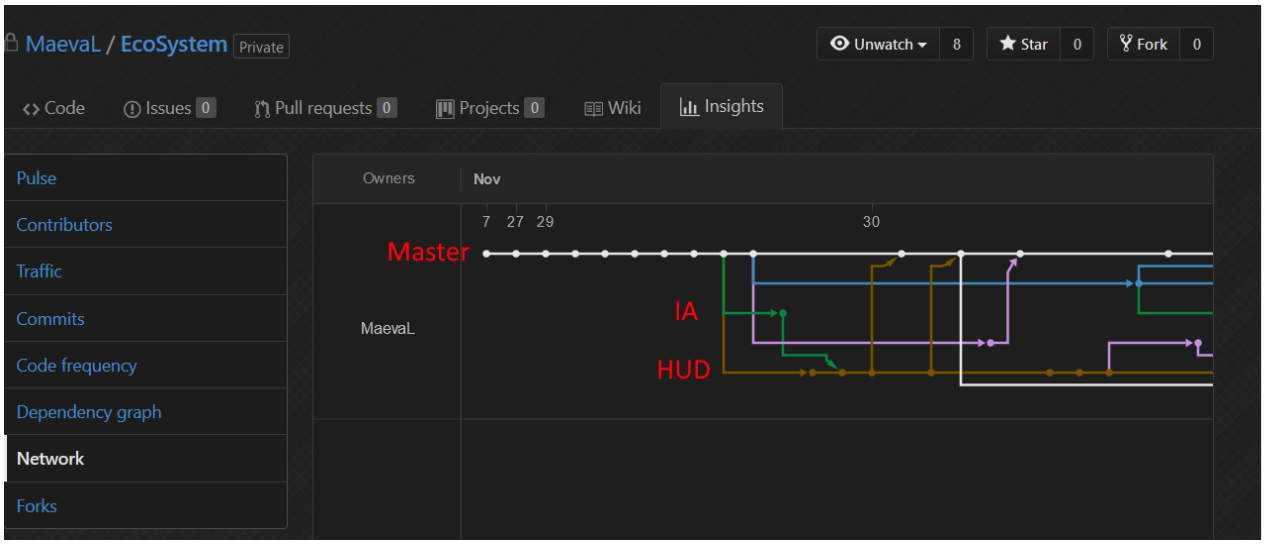
Par chance il est bien plus simple de récupérer le travail des autres que de travailler soi même. Pour recevoir les changements déposées sur le serveur (et la même branche que nous). Il suffit de taper :

1. **Git pull**
2. Enjoy !

Git c'est simple au final ? En fait non, pas toujours…..
Si vous êtes plusieurs à travailler sur une même branche, il se peut qu'il y ait des problèmes de conflit/merge dès lors que plusieurs personne travaillent sur le même fichier en utilisant la même branche.

Branches ? Conflit ? Merge ?

L'une des forces de *git* est son système de branche :



Git network, branches system

En effet travailler ensemble et partager ses modifications c'est bien. Le faire intelligemment c'est mieux. Très souvent un projet peut être séparé en de nombreux sous projets indépendants (ex : gestion de l'inventaire, gestion des bâtiments, IA, contrôles du personnage, etc). Et donc répartir un projet massif en créant une branche pour chaque sous projet possède plusieurs aspects bénéfiques :

- Possibilité d'avoir une branche Master **CONSTAMMENT** fonctionnelle.
- Si une branche fait une mauvaise manipulation, commit foireux. Cela ne bloquera pas les autres branches.
- Jean-Eude n'a pas besoin de pull les modifications du HUD a chaque fois qu'il veut push ses modifications sur l'IA si ces deux aspects sont traités sur des branches différentes.
- La possibilité de faire la « différence fichier par fichier» entre deux branches montrant l'avancement de chaque branche et permettant de repérer facilement les modifications faites par ses camarades.
- Au niveau du code on se rend compte assez rapidement que deux parties du code que l'on pensait indépendantes se retrouvent fortement liées si deux branches commencent à se marcher dessus, modifier les mêmes fichiers, etc
- etc

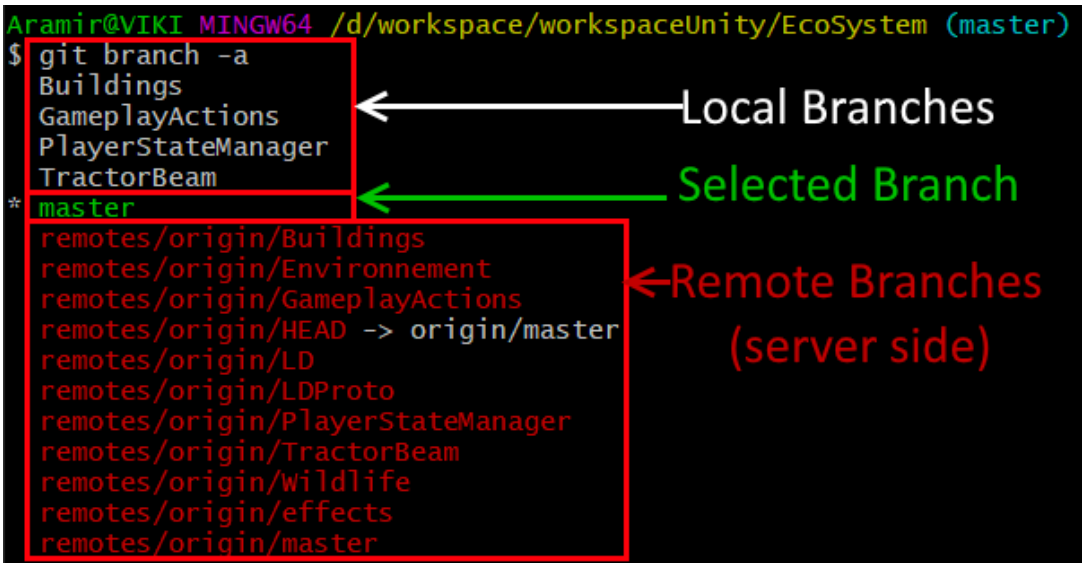
Le « workflow » habituel consiste donc à se découper le travail en plusieurs taches/fonctions indépendantes. De travailler indépendamment chacun de son côté. Et une fois son travail terminé ET VÉRIFIÉ d'appliquer les changements sur la branche du master. Le master DOIT resté le plus propre possible, c'est la branche par défaut, celle mise en avant sur le site du github, etc.

Du coup, ok les branches c'est la vie. Travailler chacun sur sa branche garantie bonheur, félicité, repousse des cheveux et retour de l'être aimé. Mais comment qu'on fait ?

Plusieurs commandes :

- **git branch -a** : va lister la totalité des branches connues par *git*.
- **git fetch** : va lister la totalité des différences de TOUTES les branches entre la version serveur et votre version local. (Définition plus qu'approximative). Qu'est ce que ca veut dire pour vous ? Si vous ne voyez pas une branche créée par quelqu'un et pushée sur le serveur en faisant « git status », faites un « git fetch » et elle devrait alors s'afficher.
- **git checkout [BRANCH_NAME]** : Permet de passer d'une branche à une autre. ATTENTION cependant, il vous faut impérativement commit (pas forcément push) toutes vos modifications avant de passer d'une branche à l'autre. (Si vous essayez un message d'erreur devrait vous le rappeler de toute manière).
- **git checkout -b [NEW_BRANCH]** : Permet de switcher sur une branche et la CRÉER si celle-ci n'existe pas. Attention cependant cela ne la créera que en local, pour que celle-ci existe côté serveur il vous faudra impérativement commit et push des modifications. Lorsque vous essaierait de push un message d'erreur devrait apparaître vous informant que la branche n'existe pas côté serveur. Celui-ci devrait vous indiquer de taper la commande suivante.
- **git push --set-upstream origin [NEW_BRANCH]** : Permet de créer une branche côté serveur et lier votre branche locale à celle-ci. De façon à ce que tout changements associés (commit/push/etc) à cette branche soient envoyés sur la branche [NEW_BRANCH] du serveur.

NB : Lorsque vous tapez « git branch » vous vous retrouverez face à un écran de la sorte. Gardez en tête que si vous n'avez pas taper « git fetch » auparavant, plusieurs branches côté serveur peuvent ne pas avoir été découvertes encore et donc être ignorées par « git branch ».



Explication branches locales, branches remote

"Ok cool, mais dans le titre y a marqué MERGE, kesako ?"

Le merge c'est l'action qui consiste à fusionner deux branches / Ramener toutes les modifications d'une branche sur une autre. Imagineons que Charle-André ait terminé d'implémenter le HUD, il souhaite donc se mettre à travailler sur la gestion de caméra. Pour se faire il va devoir "merge"/appliquer ses modifications sur le master avant de se créer une nouvelle branche "GestionCamera". Pour appliquer ses changements sur le master il va donc faire :

- **git add . / git commit / git push** : La trinité habituelle pour commit et push toutes ses modifications sur sa branche HUD.
- **git checkout master** : Se déplacer sur la branche master
- **git merge HUD -m "merging HUD onto master"** : Merger la branche HUD sur la branche courante/master et commente l'opération, comme lors d'un commit.

NB : Gardez en tête lorsque vous faites un merge que vous rammenez les changements de la branche ciblée sur la branche courante. Autre piège, on ne peut évidemment merge qu'avec les branches que le git local connaît. Pensez donc à faire un « git fetch » auparavant afin de vous assurer que votre liste de branche local soit complète. Et une fois le merge effectué (côté local donc), il vous faudra push les modifications effectuées (côté serveur). Sachez aussi qu'il est possible de merger diretement depuis des branches présentes sur le serveur mais absente en local. Pour cela la syntaxe est quasiment identique :

- **git merge origin/SomeRemoteBranch** : La seule différence étant qu'il faut préfixer le nom de la branche par « origin/ » afin d'indiquer que celle-ci se trouve sur « origin » aka le serveur git. Par défaut si aucun préfixe n'est indiqué le merge s'effectuera sur une branche locale.

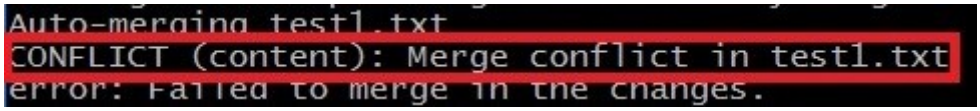
Et en géneral c'est là que tout explose... parceque les différences entre les deux branches sont en géneral espacées de plusieurs jours / semaine parceque chacun a pris des libertés sur le code de l'autre, etc. *Git* va essayer au maximum de fusionner les fichiers intelligements. Mais lorsque un doute planne il va signaler un CONFLIT et ca va être au développeur de fusionner à la main les fichiers en conflit / que *git* n'a pas su fusionné. (En géneral on laisse le sale travail de merge au développeur). C'est aussi géneralement là que la plupart des outils visuels proposent des solutions de merge automatiques... Qui ont bien souvent tendance à faire plus de mal que de bien... Cliquer sur un gros bouton sans comprendre les implications derrières jusqu'a ce qu'on puisse enfin pusher ses modifications n'apporte rien de bon sur le long terme. Le gros bouton magique, ca n'existe pas !

Ok mais concrétement, que se passe-t-il lorsque *git* doit merger deux fichiers ? Il va tout simplement faire la différence entre les deux versions (branche A, branche B) de tout les fichiers un à un. Et suivre les règles suivantes:

- Si un fichiers est modifié sur les deux branches mais que les modifications ne se chevauchent pas (exemple: Jean marc a modifié la fonction permettant d'incrémenter le score du joueur là où Pierre Antoine a touché à la fonction d'affichage du score), alors *git* appliquera les deux modifications sans sourciller car il s'agit de parties différentes du code.
- Si deux modifications se chevauchent alors il FAUT que ce soit les mêmes modifications dans les deux branches. Autrement une alerte de CONFLIT sera lancé et le développeur devra corriger le conflit à la main
- Les modifications telles que le saut de ligne, le rajout d'espaces sont géneralement ignorés.

Pour les curieux il est possible de faire un *git merge recursive*, dans ce cas il va :

1. Chercher une base commune pour les deux fichiers en regardant dans les commits précédents
2. Appliquer successivement les commits des deux branches selon l'ordre dans lequels ils sont arrivés (en effet chaque commit contient sa date d'application ce qui permet d'appliquer les commits de différentes branches dans l'ordre)
3. Faire la vérification habituelle à chaque application d'une différence.



Message d'erreur typique lié à un merge / push

"Ok super les conflits c'est vilain comme elle disait ma maman. Mais comment qu'on gère ça?"

Vous ne le gérez pas. Non vraiment.... Pour gérer un conflit il faut être au courant des changements effectués sur les deux branches, savoir quelle modification a été apportée, pourquoi, analyser, identifier, corriger le code... Bref c'est pas forcément évident. Mais si jamais, en cas d'extreme urgence, les devs se sont tous fait kidnappés par Mickey et sont retenus prisonniers à disneyland.

Voici comment résoudre les conflits à la main.
Il suffit d'ouvrir les fichiers signalés dans la console comme posant des problèmes de conflits.
Normalement ceux-ci devraient alors contenir les modifications des deux branches de la façon suivante :

```
12      public void quit()
13      {
14  <<<<<<< HEAD
15          Debug.Log("1+1= "+(1+2)); "HEAD" SPECIFIC CODE
16  =======
17          Debug.Log("1+1= "+(1+1));"Commit we're trying to merge" SPECIFIC CODE
18  >>>>>>> Some commit Message or Commit id (65168265365151151f351fc1da16ca1a651cca)
19          Application.Quit();
20      }
```

Conflict in code

Entre <<<<< HEAD et ======= se situe le code de référence / le code contenu dans la branche actuelle.
et entre ======= et >>>>>>> Some commit Message sont contenus les changements apportés par le commit que l'on essaie d'appliquer / la branche que l'on essaie de merger.

Stash ? Pop ? .gitignore and other Tricks

Bon alors là mettons nous d'accord. Je vais évoquer rapidement quelques commandes supplémentaires dont vous aurez en théorie jamais besoin. Mais sait-on jamais ça peut vous être utile histoire de pas passer pour une tanche auprès de votre programmeur préféré. Et je préfère que vous les connaissiez vaguement avant de les copier coller depuis StackOverflow sans comprendre.

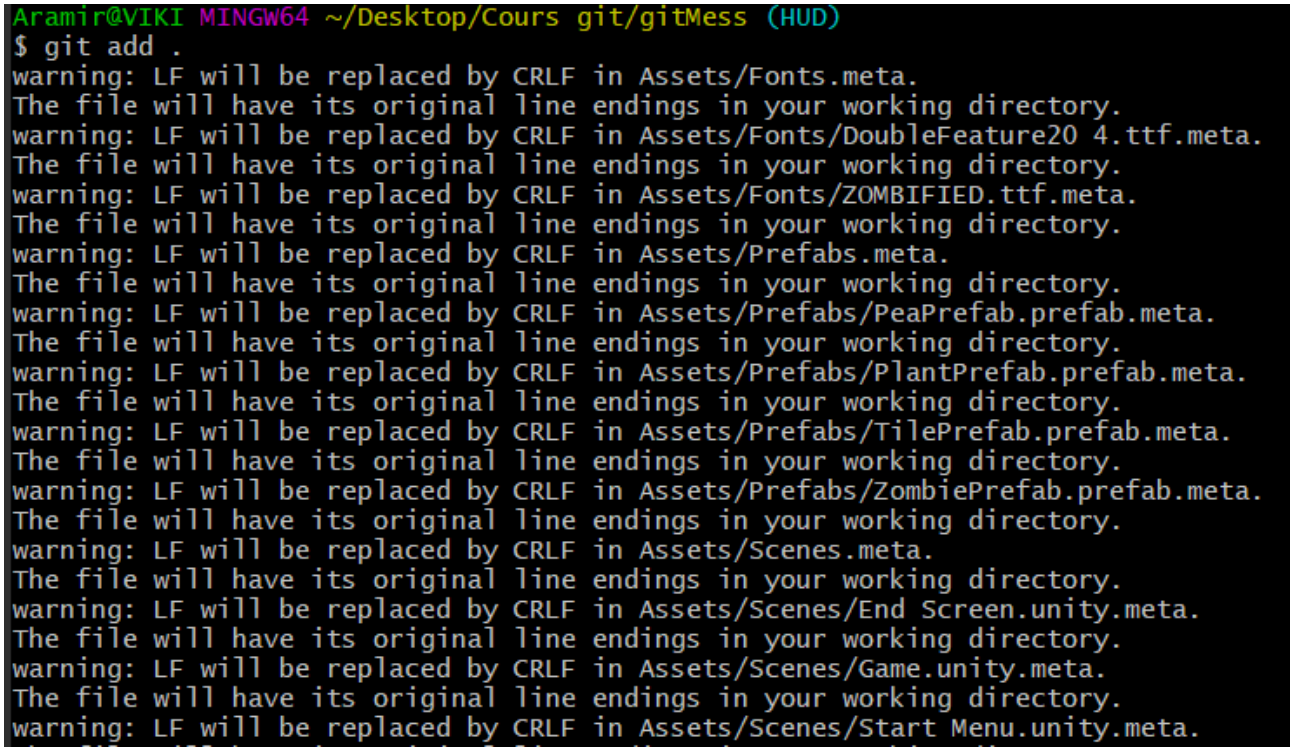
- **git reset --hard HEAD** : L'Armageddon de *git*, lorsque vous avez fait une mauvaise manipulation et que vous souhaitez annuler toutes vos modifications, que vous voulez revenir en arrière. Utilisez cette commande pour revenir à la dernière version que vous avez commit (en local). « git reset –hard HEAD~2 » reviendra deux commit en arrière.
- **git stash** : permet de mettre toutes les modifications effectuées depuis le dernier commit dans une « branche locale cachée » et revient automatiquement à la version du commit précédent. Utile lorsque l'on veut résoudre un merge et qu'il est inutile de garder les modifications que l'on a effectué (parcequ'on a merdé quelque part, parce que le collègue a fait tout le taf à ta place et du coup tu peux jeter ton code tout pouri, etc). On garde donc toutes nos modifications dans un coin « au cas où » et on passe à la suite. Cela peut aussi avoir de véritables applications comme lorsqu'un développeur doit basculer rapidement d'une tache à une autre et souhaite garder les modifications en local afin d'y revenir plus tard.
- **git pop** : permet d'appliquer les modifications stockées précédemment grâce à *git* pop. Utile lorsqu'on se rend compte que « finalement le code que j'ai fait été pas si con que ça, j'ai bien fait de le garder quelque part »
- **.gitignore** : Ce n'est pas une commande mais un fichier caché. Il permet d'éviter d'envoyer certains fichiers inutiles sur le serveur (ex : les builds qui prennent de la place, la config de Thierry qui code sur un clavier en braille, les fichiers temporaires de Unity, etc)
  Il se construit de la sorte :
  - un dossier/fichier à ignorer par ligne, ex :
    - Library/ : ignore le dossier Library
    - [Ww]indowsBuild/ : ignore les dossiers windowsBuild et WindowsBuild
    - /*.user : ignore tout les fichiers à la racine du dossier avec l'extension .user
  - Les lignes commençant par # sont des commentaires.
  - Les lignes commençant par ! permettent de forcer la prise en compte de certains fichiers dossiers, même si ceux ci se trouvent dans des dossiers ignorés par une ligne précédente, ex :
    - !Library/BuildSettings.asset
    - !Library/BuildPlayer.prefs
- **git diff** : Affichera toutes les différences entre vos fichiers et les fichiers disponibles sur le serveur (pour votre branche). Possibilité de raffiner l'affichage en n'affichant les différences que pour un fichier (git diff [FILE]).
- **git commit –amend -m « new commit message »** : Redéfinit le message du commit précédent. Au cas ou par fatigue/énervement/etc tu es insulté toutes les races de l'univers dans ton commentaire de commit précédent et tu ne veux pas que ça se sache.
- **git checkout –theirs [filename]** : Lors d'un conflit, il peut arriver que l'on ne souhaite pas trier les différences à la main mais que l'on souhaite juste récupérer le fichier présent sur le serveur en ignorant totalement nos modifications. C'est ce que fait « git checkout –theirs [filename] », il va chercher le fichier « filename » présent sur (la branche du) le serveur et le remplacer notre fichier tel quel par celui-ci. Écrasant ainsi toutes nos modifications.
- **git checkout –ours [filename]** : identique à la différence prêt que l'on conserve notre fichier.

Annexe, Troubleshootings

AKA « la rubrique a consulté avant d'aller pleurer chez les devs »

- Je comprends pas j'ai tapé « git checkout -b [Controllers] » et ca marche pô :
  Alors le coup de mettre des [ ] dans les textes que j'écris est une convention utilisée pour signaler une variable. Autrement dit essaie de taper « git checkout -b Controllers » sans les [] ca devrait aller.

- « J'ai plein d'erreurs au moment d'add mes fichiers » :

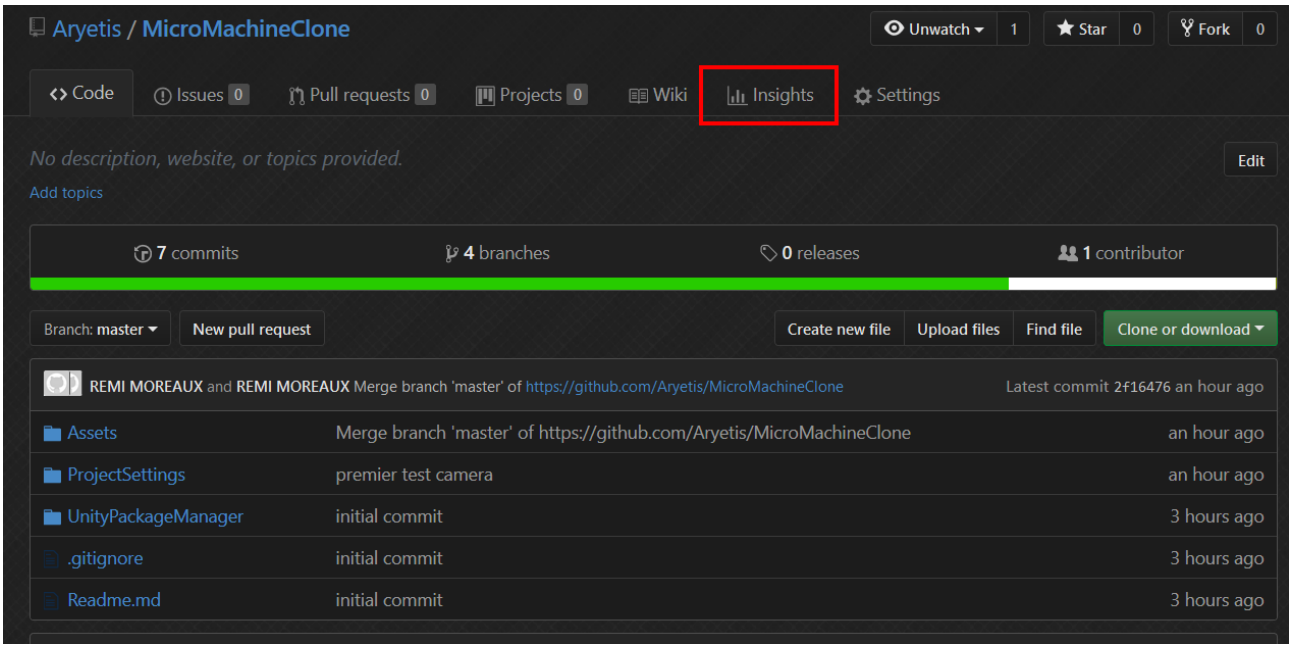


Erreurs fins de lignes Linux

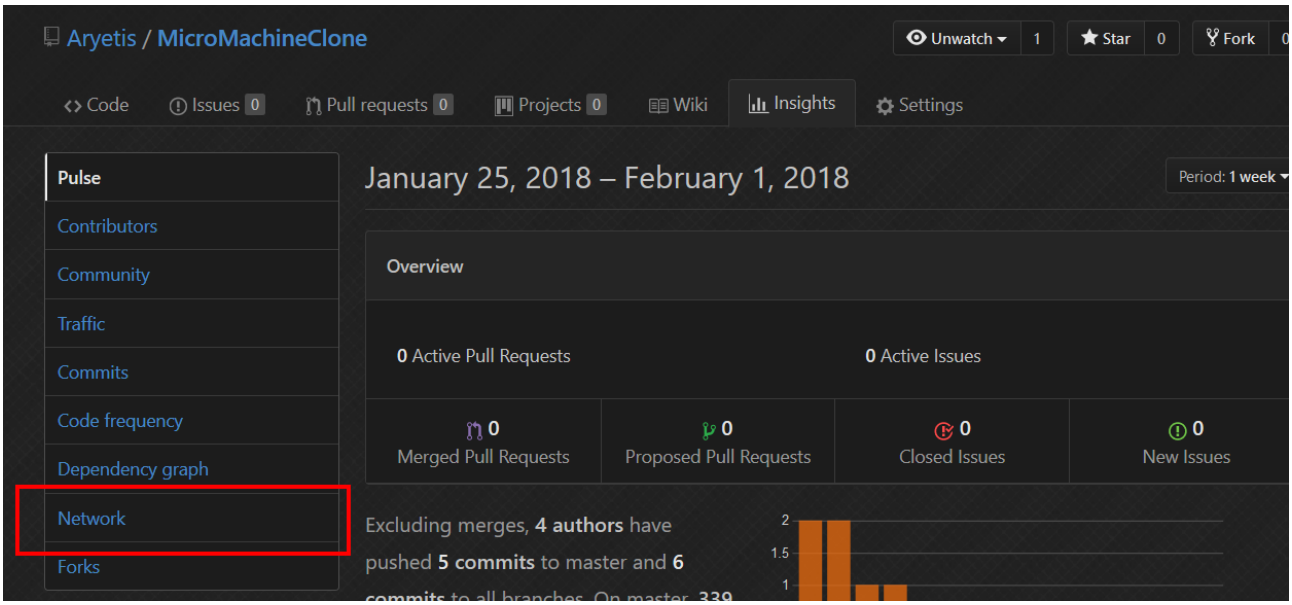
Félicitations tu viens de trouver la seule erreur sans aucune importance. Tu peux l'ignorer sans aucun soucis. Elle indique simplement que les fins de lignes sont codés au format « Windows » en local et format « Unix » sur le serveur. Comme indiqué, *git* les remplacera automatiquement. Pas de soucis à se faire.

- « Je comprend pas je trouve pas le .gitignore » : Par défaut windows cache les fichiers sans noms / démarrant par un '.' et cache les extensions de fichiers. Une simple recherche google devrait te permettre de trouver comment afficher les fichiers cachés et extensions. Tadaaaa.

- « Comment visualiser les branches présentes sur le serveur avec leurs commits associés comme t'as montré à la page 14 ? Ça à l'air pratique. ». Ça l'est et il suffit de se rendre sur la page principale de son Repo Github puis dans l'onglet Insight → Networks



Github onglet insight



Github onglet Network

- « Comment qu'on fait pour supprimer une branche ? » : **git branch -d [BranchName]**

- À vrai dire je peux pas imaginer d'autres erreurs là dans l'immédiat donc si vous en avez faites passer et je les ajouterai au fur et à mesure….