# Gestion des projets

① ② ③ ④ ⑤

Dev → Alpha → Beta → RC (release Candidats) → Release   
 1ers tests + grand nombre testeurs sur le marché version du marché  
 Bugs de testeurs version vendue

Versions :

↗ corrections + tests

1. → 1.1 → 1.2→Mise à jours

↘ 2.0→ 2.1 → 2.2  
 ↘nouvelles fonctionnalités → 3.0

→ suivre le code source et les modifications  
→ Retrouver les fichiers effacés  
→ revenir en arrière  
→ bugs de régression

# Gestion des versions

① Local :

Semaine 1 → 🗀 sem1  
Semaine 2 → 🗀 sem2

② C. VCS : (Central Versionning Central System)  
ex : SVN, CVS (central de stockage des fichiers)

③ D. VCS: (Distributed VCS)

Fichier partagé avec tous les utilisateurs

Ex : GIT (linux)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Untracked  Fichier non suivi |  | Unmodified  Non modifié |  | Staged  Indexé |  | validé |
|  |  | Git add\_. |  | Git commit |  |  |

Git.scm.com (64 bits win) A télécharger et installer

Cocher les 2 cases sous windows intégration

Pour la gestion des projets

# GIT BASH

Commandes :

Mkdir␣projet01

cd␣projet01/

cd␣..(remonte le fichier)

mv␣projet01␣Desktop/ (positionne le fichier sur le bureau)

git␣init

git␣add␣.

Commandes :

Mkdir créer un répertoire

cd changer de répertoire

cd␣.. remonter de répertoire/répertoire parent

cd. répertoire courant

Ls liste des fichiers

cat␣file afficher le contenu (file = nom du fichier)

pwd afficher le nom du répertoire courant

git␣clone : url http ou https ou git://

vi créer un fichier

git␣status voir le statut

git␣config␣- - global ␣user.name= « ␣ » configuration système pour l’utilisateur

git␣config␣- - global ␣user.name= « ␣ »

git␣config␣ -- list

git␣config␣--global entrer des infos

git␣add+nom de fichier ajoute un fichier

git␣commit valide un fichier

git␣commit␣-m « ␣ » ajouter un commentaire à la validation

git␣commit␣-a␣-m git␣add+git␣commit en même temps

git␣log montrer ce qui est validé

git␣log␣-- online

git␣diff différence entre 2 fichiers

git␣checkout␣fichier permet d’extraire un fichier de la base de données → passe d’une version à une autre

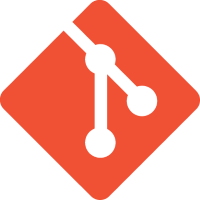
git␣branch liste des branches dispo du projet (la branche active est marquée d’une étoile)

git␣branch␣-v voir les dernières modifications

git␣branch␣-m␣fichier

ll␣ls.l listing complet des fichiers

git␣push␣-u␣origin␣master transfère le dossier complet sur le serveur



**1. Qu’est-ce que Git ? Les concepts à comprendre**

Git est un **logiciel de gestion de versions distribué**, créé par [Linus Torwalds](http://fr.wikipedia.org/wiki/Linus_Torvalds)  en 2005 et distribué gratuitement sous **licence publique générale GNU version 2**. Si la fin de la phrase est simple à comprendre (quoi que…), nous allons détailler son début :

**Un outil de gestion de versions**

Le but de Git est de permettre à une équipe de développement de **suivre les différentes versions d’un projet** pour en gérer l’évolution. Oublions un peu le code et imaginons un exposé sur l’économie du cacao en Côte d’Ivoire réalisé par 3 valeureux lycéens : Pierre est en charge de l’exposé en lui-même, Paul rédigera l’introduction et la conclusion, Jacques aura la responsabilité des illustrations et de la mise en page. Pierre enregistre son document sur une clé USB, qu’il transmet à Paul. Paul ajoute son introduction et sa conclusion sans relire le travail de Pierre puisqu’il lui fait confiance. Il donne ensuite la clé USB à Jacques qui fait son travail de mise en forme avant de donner la clé USB au professeur. Le professeur consulte le document et s’aperçoit qu’il manque un chapitre entier, celui consacré au commerce équitable. Qui est fautif ? (vous avez 3 heures…) Git est là pour répondre à cette question en permettant de **garder une copie de toutes les versions** des documents, avec pour chaque version, l’identité de **la personne qui a fait les modifications** et les **détails de cette modification**. Appliquez cela à n’importe quel projet, informatique ou non, et vous aurez compris l’intérêt de Git.

**Un logiciel distribué**

Peu importe le nombre de machines qui travaillent sur le même projet, chacune d’entre elles possède **une copie intégrale du «dépôt»** (nous expliquerons le terme par la suite) et est capable de travailler seule pour partager ses mises à jour en temps voulu avec les autres intervenants. Cela a un impact en terme de **sécurité**, puisque sans serveur central, la **tolérance aux pannes** et plus grande : il suffit qu’une seule machine soit en vie pour que le projet puisse être redistribué à volonté. Cela a aussi un impact en terme de **performances** : si on ne fait pas d’accès réseau en permanence on gagne forcément du temps. Le revers de la médaille est que des **conflits** peuvent survenir si plusieurs personnes modifient les mêmes lignes de code chacune de leur côté.

**2. Le vocabulaire de Git**

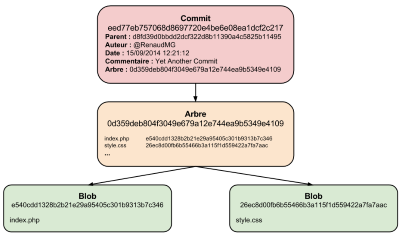
Nous allons utiliser plusieurs mots que vous ne connaissez peut-être pas, ce chapitre est donc un lexique auquel vous pourrez vous référer lors de votre lecture

**dépôt, repository ou remote**

Je vais encore oser une métaphore simpliste : quand vous mangez, vous utilisez des couverts. Lorsque vous avez terminé votre repas vous lavez ces couverts et vous les rangez dans un tiroir où d’autres pourront les utiliser. Un dépôt peut être comparé à ce tiroir (si si, il faut nettoyer son code avant de le ranger, tout comme une fourchette). Il s’agit d’un endroit où l’on va stocker les versions du code, à disposition de tous les participants au projet qui vont venir en faire une copie chez eux et éventuellement faire des mises à jour. On me dit à l’oreille qu’on ne peut pas faire une copie des fourchettes du tiroir, effectivement ma métaphore a un défaut. Un projet peut avoir un ou une multitude de dépôts, en lecture seule ou sur lesquels l’écriture est permise… c’est vrai que c’est plus compliqué que le stockage de l’argenterie mais je pense que vous avez compris l’idée.

**Validation ou commit**

Un commit n’est ni plus ni moins qu’une version du projet, un instantané pris à un instant T. A chaque fois que vous faites un commit, Git enregistre un «arbre» qui permettra d’identifier vos fichiers modifiés (qui seront stockés sous forme de blobs) et lui associe un objet commit qui, lui-même, pointera vers le (ou les) commit immédiatement précédent.

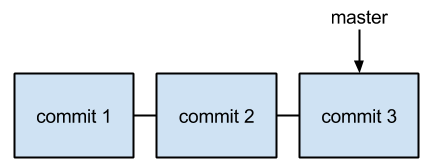


Représentation d’une validation Git

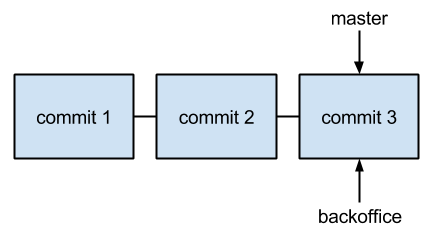
Dans l’esprit du projet, un commit devrait correspondre à la mise en ligne d’un morceau de code testé et validé. Ca n’est pas toujours le cas, parce qu’on peut faire un commit sur une branche qui n’est pas destinée à la production.

**Branche**

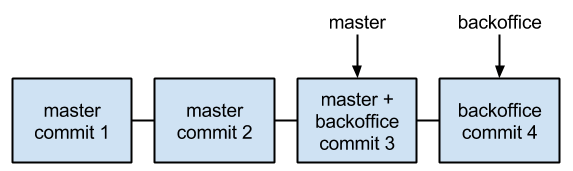
La notion de branche est certainement la plus difficile à comprendre dans Git. Nous venons de voir ce qu’est un commit, une branche est simplement un pointeur vers un commit. Puisque chaque commit connaît son précédent, il est possible de remonter de commit en commit pour retracer toute la branche. Pour simplifier, imaginons que nous avons créé un projet et fait 3 commits. Alors que nous ne nous sommes pas souciés de créer une branche, Git l’a fait pour nous. Dès le début le projet possède une branche par défaut, appelée master. Après nos 3 commits, master pointe vers le dernier commit en date.



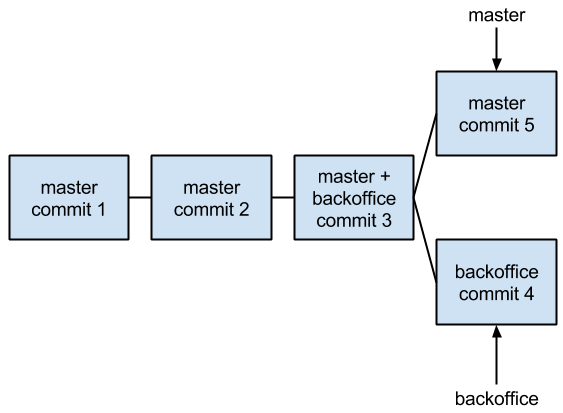
Nous souhaitons maintenant créer une branche pour permettre à l’équipe back-office de travailler de son côté, Git va simplement créer un nouveau pointeur vers le dernier commit de la branche courante (donc master). Nos branches master et backoffice sont pour le moment identiques



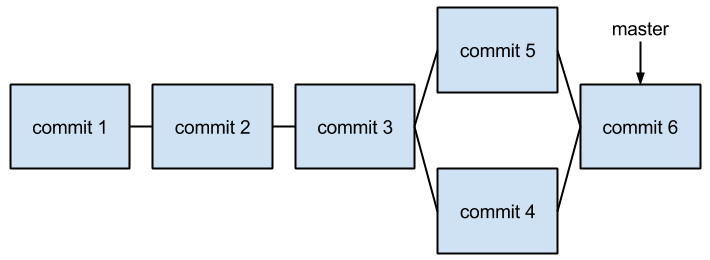
Si nous activons la branche backoffice et que nous faisons à nouveau un commit, le pointeur de la branche backoffice va se déplacer pour pointer sur le nouveau commit tandis que le pointeur de master ne bougera pas.



Si nous nous plaçons à nouveau sur master et que nous faisons encore un commit, master va venir pointer vers ce dernier et nous aurons **deux branches parallèles**.



Je vous ai dit qu’un commit pouvait faire référence à plusieurs commit précédents, c’est le cas lorsque l’on **fusionne deux branches**, dans notre exemple la fusion de backoffice  dans master entraînera la création d’un nouveau commit dont les «parents» sont à la fois c4 et c5.



**3. Initialiser un dépôt Git**

Comme nous l’avons vu, un projet Git peut concerner tout type de documents et nous n’aurons donc aucune contrainte pour initialiser un projet, ce que l’on appelle aussi **créer un dépôt**. Il suffit de se placer dans le dossier qui va contenir le repository et de taper la commande

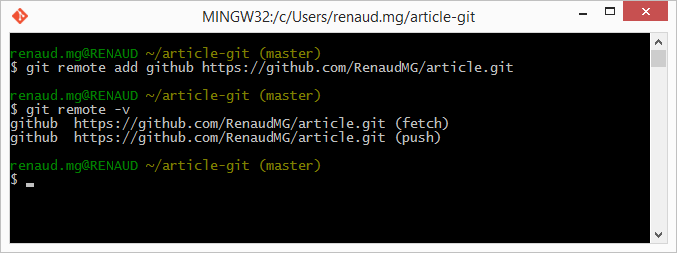
git init

Votre dépôt est maintenant prêt à fonctionner, vous pouvez voir qu’un dossier caché nommé «.git» a été créé dans votre répertoire.

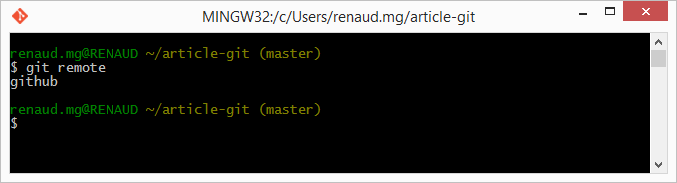
**4. Se connecter à un dépôt**

Je vous rappelle que l’intérêt de Git est le suivi des modifications, mais aussi la **sauvegarde de vos projets**, c’est pourquoi je vous conseille de toujours commencer par **copier vos sources sur un dépôt distant**, si possible situé à l’extérieur de vos locaux (je vous épargne le débat stérile sur la paranoïa des DSI, j’ai connu l’incendie d’une salle serveurs lorsque j’étais étudiant et c’est seulement après cet incendie que nous avons réalisé qu’il n’était pas judicieux d’entreposer les bandes de sauvegarde dans un carton posé à cheval sur les serveurs…).  
Pour cet exemple je vous propose donc d’utiliser Github, qui est un serveur Git sur lequel chacun peut créer, stocker et administrer ses dépôts en fonction de l’abonnement qui est souscrit. Nous n’allons pas faire le tour du propriétaire maintenant, cela serait bien trop long pour cet article, nous allons nous contenter de connecter notre projet à ce service.

git remote add nom\_du\_depot url\_du\_depot



Encore une fois, c’est très simple, seuls deux paramètres sont obligatoires : en premier le nom à donner à notre remote, en second son url. Notre projet est donc maintenant connecté au remote «github». Si votre projet provient du **clonage d’un remote** existant (nous y viendrons plus tard dans cet article), ce dépôt aura automatiquement comme nom “**origin”**.  
Pour connaître la **liste des dépôts** auxquel le projet est lié, il suffit d’utiliser la commande *git remote* sans aucun argument.

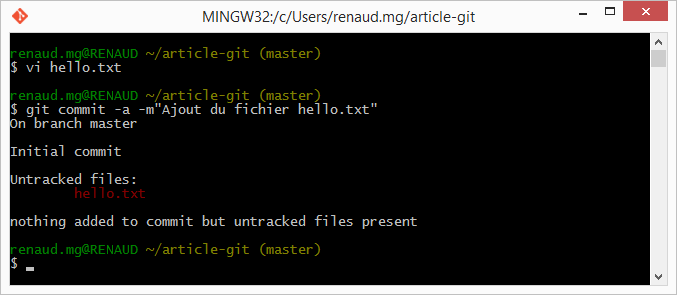


**5. Ajouter un fichier au projet Git, supprimer ou renommer un fichier**

Nous allons créer un petit fichier «hello.txt» dans le répertoire de travail et faire un commit pour valider cette version du projet. Nous aborderons le commit dans le prochain paragraphe, donc ne vous focalisez pas sur la commande pour l’instant, c’est son résultat qui nous intéresse.

git commit -a -m"Ajout du fichier hello.txt"

Comme vous pouvez le voir dans la capture ci-dessous, Git nous répond “*nothing added to commit but untracked files present*“.

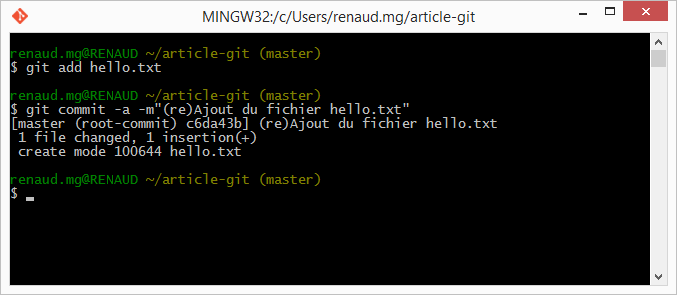


ce qui signifie que le fichier hello.txt **ne fait pas partie du projet** et qu’il n’a pas été pris en compte dans la validation.  
Pour remédier à cela nous allons **explicitement ajouter notre fichier** au projet avec la commande

git add hello.txt

Si on réessaie de faire un commit, le résultat est différent :

git commit -a -m«(re)Ajout du fichier hello.txt»

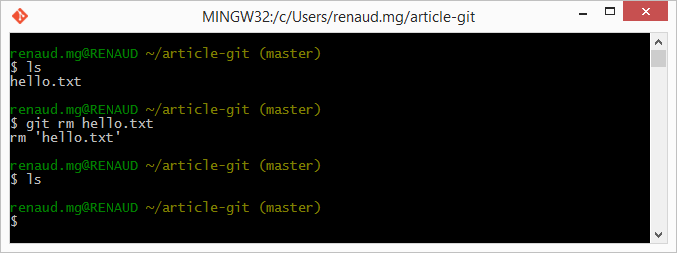


Le fichier est bien pris en compte et a été tracké correctement. Ici nous n’avons ajouté qu’un fichier, mais on aurait très bien pu ajouter tous les fichiers texte (*git add \*.txt*) ou tous les fichiers (*git add \**).

Soyez vigilants vis à vis de cette étape : lorsque l’on débute avec Git il est fréquent d’oublier l’ajout de fichiers avant un commit et on a parfois quelques surprises. Par exemple dans le cas présent, si je ne suis pas attentif au message du premier commit je ne verrai pas que le fichier n’a pas été ajouté correctement. Si je me positionne sur ce commit, je ne retrouverai pas le fichier hello.txt annoncé.

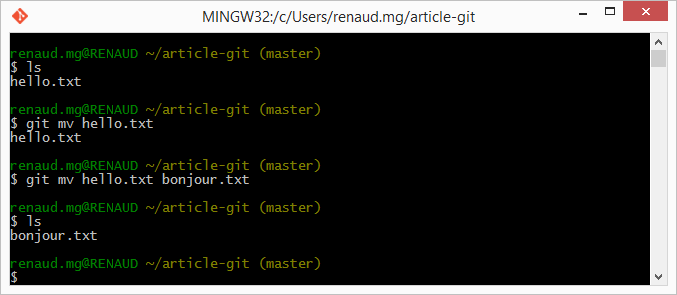
Pour **effacer un fichier** la commande est :

git rm hello.txt



et pour **renommer ou déplacer un fichier**, il faut utiliser :

git mv ancien\_nom nouveau\_nom

****

**6. Valider une version du projet**

Tout comme l’insupportable gamin qui ordonne sans cesse à Shrek «fais ton greuh», le chef de projet répète à chaque milestone aux développeurs «fais ton commit» (ainsi que «fais un push après ton commit», «tu as fais un fetch ?» …).  
Le commit est l’action d’**enregistrer une modification** sur le projet : j’ai écris mon code, ce code a été testé et validé, je fais un commit pour créer une nouvelle version du projet et mettre mon travail à disposition des autres acteurs.  
Chaque commit est **accompagné d’un message** qui se doit d’être **explicite** pour pouvoir comprendre d’un simple coup d’oeil ce qui a été mis à jour. Ce commentaire est introduit par l’argument -m de la ligne de commande, suivi du commentaire entre guillemets. Attention, il n’y a pas d’espace entre -m et les guillemets. Vous devez aussi savoir que si vous ne spécifiez pas de commentaire, Git ouvrira un éditeur de texte pour vous forcer à le saisir. Si vous vous entêtez à ne rien saisir, le commit ne sera pas fait.  
Il est possible de sélectionner un seul fichier à inclure au commit mais la plupart du temps on se contente de valider tous les fichiers, ce qui est signalé par le paramètre -a  
Vous comprenez maintenant la commande que nous avons utilisé au paragraphe précédent, à savoir :

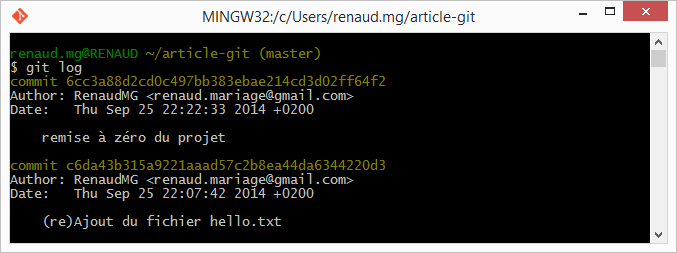
git commit -a -m"message\_de\_validation"

**7. Visualiser l’historique des validations**

Il est possible de visualiser l’**historique du projet** d’un simple coup d’oeil grâce à la commande

git log

qui va nous afficher une liste ressemblant à celle-ci :



Pour chaque commit nous pouvons lire :

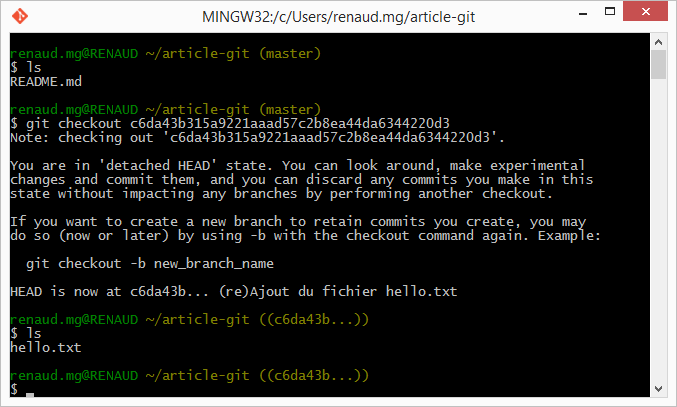
* Une étrange chaîne de caractère, le sha1, qui sert d’**identifiant à la validation** comme nous l’avons vu au tout début de l’article
* La **date de la validation**
* Son **auteur**
* Le **message** qui lui a été associé, vous comprenez maintenant pourquoi il est nécessaire d’y prêter attention lorsque vous validez votre code

**8. Se positionner sur un ancien commit**

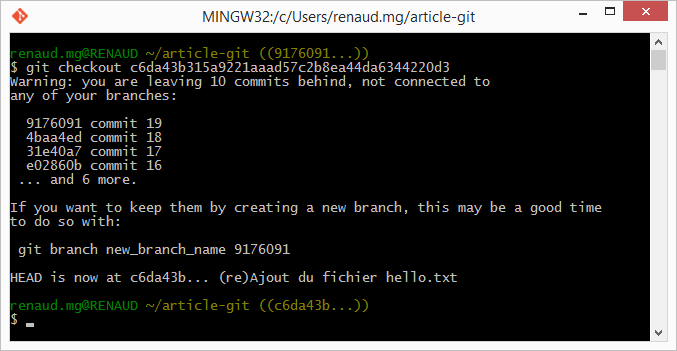
Si pour une raison ou une autre vous désirez revenir en arrière, sur votre validation précédente ou même 10 commits dans le passé, il suffit de noter l’identifiant du commit dans l’historique et d’utiliser la command git checkout pour s’y placer :

git checkout sha1\_du\_commit

Git se chargera de **remettre votre projet dans l’état exact** dans lequel il se trouvait à ce moment précis. Il vous suffira de créer une nouvelle branche à partir de ce point pour travailler.



**Attention :** il est plus que conseillé de créer une nouvelle branche à partir de ce point, sinon toutes les validations qui existaient jusqu’à maintenant seraient détachées de la branche courante. Imaginons que vous ayez fait des commits 10, 11, 12 … 19. Vous revenez sur le commit 10 et voulez créer le commit 20 sans faire une nouvelle branche, cela aurait pour effet de détacher les commits de 11 à 19, et c’est donc très dangereux.



**9. Envoyer son projet sur un dépôt distant**

Ca n’est pas parce que vous avez fait un commit que tous les participants au projet y ont accès instantanément. Il va d’abord falloir que vous **envoyiez votre commit au(x) remote(s)** sur le(s)quel vous êtes connecté. Comme toujours, la commande est très simple :

git push nom\_du\_depot nom\_de\_la\_branche

Pour notre exemple nous allons pousser la branche «master» sur le dépôt «github». Il est possible que Git vous demande votre nom d’utilisateur et votre mot de passe sur le repository si celui-ci l’exige, ce qui est le cas pour notre exemple.



**10. Mettre à jour son projet à partir d’un dépôt**

En fait c’est ça la phrase que l’on répète le plus, surtout si les développeurs n’ont pas encore l’habitude de travailler avec Git : “*tu as fait un fetch ?*“. Puisque chacun est succeptible d’apporter des modifications au projet il est indispensable que tout le monde dispose en permanence du code le plus récent et donc que tout le monde **demande fréquemment au dépôt principal s’il y a des mises à jours**. Cela se fait avec la commande

git fetch nom\_du\_depot

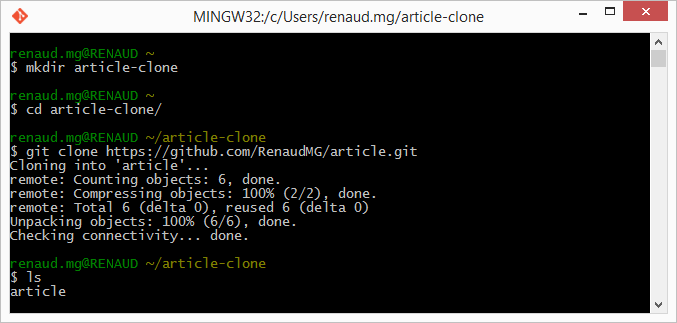
Chaque modification validée sur le dépôt sera alors effectuée sur votre copie du projet, sauf en cas de conflit, par exemple si vous et un autre de vos collaborateurs avez travaillé sur la même ligne du même fichier.  
Vous devez aussi savoir que si vous souhaitez faire un push alors que vous n’êtes pas à jour par rapport au remote, vous serez contraint de faire un fetch avant de pouvoir envoyer vos fichiers.

**11. Cloner un dépôt**

Dans les paragraphes précédents nous avons crée un projet à partir de la console et nous l’avons envoyé sur un serveur distant. Nos collaborateurs qui souhaitent obtenir une copie du projet n’auront qu’à le **cloner à partir du dépôt** avec la commande

git clone url\_du\_depot

qui va **ajouter un dépôt** à votre projet et en faire une **copie intégrale** sur votre machine. Ce dépôt sera nommé «origin». Si vous souhaitez vous joindre à un projet, vous n’avez rien d’autre à faire.



**12. Créer une branche et se positionner dessus**

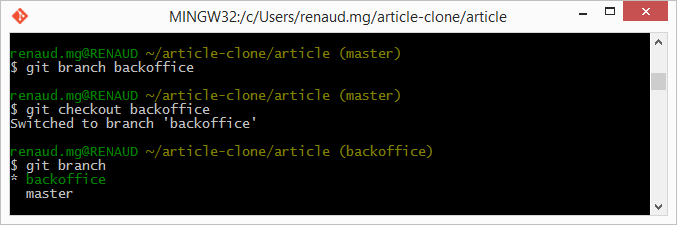
Je pense avoir bien expliqué le concept de branche (même pour les 2 au fond de la salle ?) mais je ne vous ai pas encore expliqué comment en créer une. Il suffit d’utiliser la commande

git branch nom\_de\_la\_branche

pour **créer la branche** et la commande

git checkout nom\_de\_la\_branche

pour **se positionner dessus**, il s’agit bien de la même commande que celle qui permet de se positionner sur un commit.  
En utilisant la commande *git branch* **sans aucun argument**, vous obtiendrez la **liste des branches** de votre projet, avec une “\*” devant la branche active.



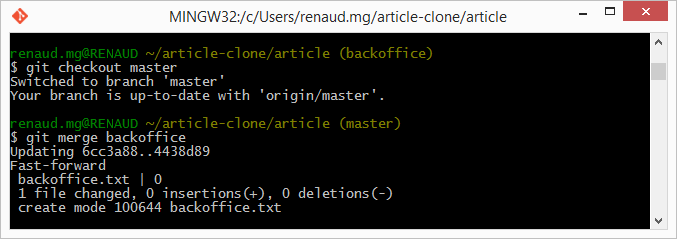
**13. Fusionner deux branches**

Reprenons notre exemple du début, à savoir une équipe de développement dédiée au backoffice. Eh bien figurez-vous qu’ils ont bien travaillé et qu’ils veulent intégrer leur travail au tronc commun du projet. Je parle de tronc commun parce que pendant ce temps, une équipe finalisait la partie utilisateurs et une troisième se concentrait sur la gestion des publications, [second degré] c’est le problème avec les méthodes Agile : tout le monde part dans tous les sens, on ne sait plus qui fait quoi ! [/second degré]. Chaque équipe a donc sa branche et dans ces équipes, chaque développeur peut avoir sa branche issue de la branche “fonctionnalité”. Le point commun entre toutes ces branches reste master, le tronc commun d’où proviennent toutes les branches “fonctionnalité” et pour faire avancer notre projet **chaque fonctionnalité testée** et validée va être **intégrée à ce tronc** commun, à l’aide de la commande *git merge*qui s’utilise en deux temps : tout d’abord on se **positionne sur la branche dans laquelle on veut faire la mise à jour**, ensuite on y **rattache l’autre branche**. Dans notre cas nous ferons donc :

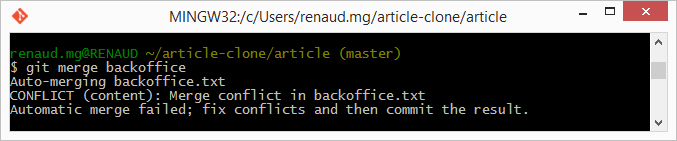
git checkout master

git merge backoffice

et toutes les nouveautés (ajouts, modification et même suppressions de fichiers) de la branche backoffice seront répercutées sur la branche master, sous la forme d’un nouveau commit, appelé **commit de fusion**… en théorie cela se passe comme ça.



En pratique c’est souvent un peu plus compliqué que cela parce qu’il arrive fréquemment qu’un **conflit** intervienne entre les deux branches, par exemple parce que la même ligne du même fichier a été modifiée sur les deux branches. Dans ce cas Git va indiquer qu’un conflit est survenu dans tel(s) ou tel(s) fichier(s) et vous devrez choisir quelle(s) modification(s) garder, puis faire un commit afin de finaliser le merge.

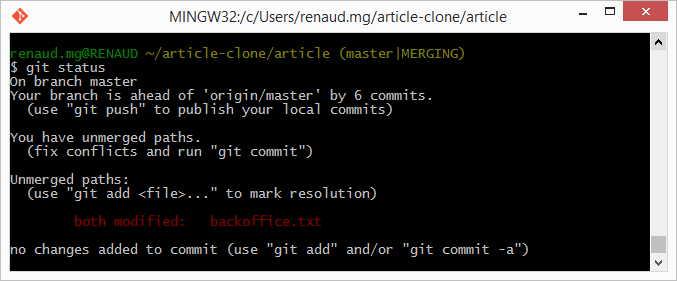


**Résoudre un conflit de fusion**

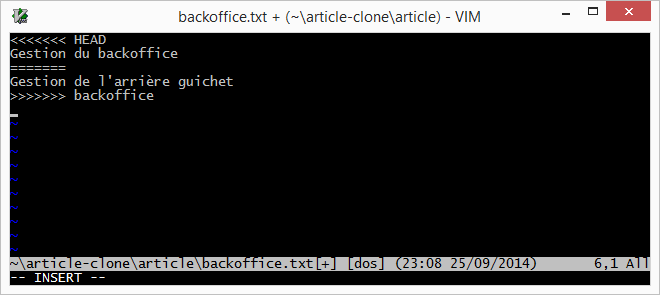
Cela peut sembler un peu compliqué de prime abord mais Git va nous aider dans la **résolution des conflits**. Voyons comment faire.  
Tout d’abord, nous allons utiliser la commande

git status

pour **savoir quels fichiers sont en conflit** : tous les fichiers qui comportent un conflit de fusion sont marqués comme **unmerged**.



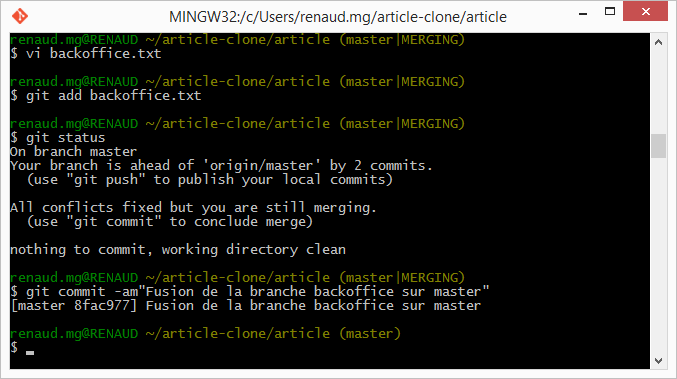
Ici c’est le fichier backoffice.txt qui pose problème, nous allons l’ouvrir et constater qu’il ressemble à ceci :



Vous pouvez facilement deviner que la section entre les <<<<<<< et les >>>>>>> présente les **deux versions en conflit**, séparées par les =======. A vous de **choisir quelle version vous souhaitez garder**, ou de tout effacer, ou de remplacer la section par ce que vous souhaitez.  
Pour indiquer à Git que nous avons **résolu le conflit**, nous utilisons la commande *git add*

git add nom\_du\_fichier

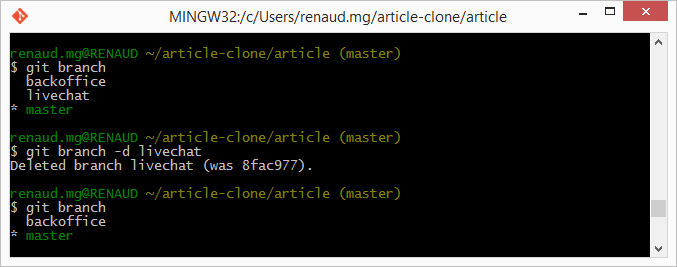
Utilisons une nouvelle fois *git status* pour vérifier que tous les conflits ont été résolus.  
Il ne nous reste plus qu’à faire notre **commit de fusion**.



**14. Effacer une branche**

Pour ne pas poluer le projet il peut être judicieux de **supprimer les vielles branches** qui ont été fusionnées de longue date, ou encore celles sur lesquelles aucun commit valide n’a pu être fait (oui on a licencié la team “live chat”, ils ont fait un peu n’importe quoi). Une simple ligne de commande et le tour est joué.

git branch -d nom\_de\_la\_branche

****

**15. Exclure des fichiers du projet**

Nous finirons notre découverte de git par le fichier **.gitignore**, qui est un simple **fichier texte** dans lequel nous allons**lister les fichiers ou types de fichier qui ne doivent pas être suivis par Git**, par exemple les fichiers de configuration, le dossier contenant les plannings de l’équipe …

# Fichiers à ne pas synchroniser

\*.doc

!readme.doc

plannings/

La première ligne de notre fichier débute par un #, c’est un commentaire et elle est donc ignorée.  
La deuxième ligne indique à Git qu’il ne doit traiter aucun fichier .doc  
La troisième ligne indique une exception à la règle précédente pour le fichier readme.doc  
La quatrième ligne demande d’ignorer tout le répertoire plannings

[separateur](http://www.responsive-mind.fr/wp-content/uploads/2013/02/separateur.png)

Source : http://www.responsive-mind.fr/git-15-points/