Petit tuto *git*

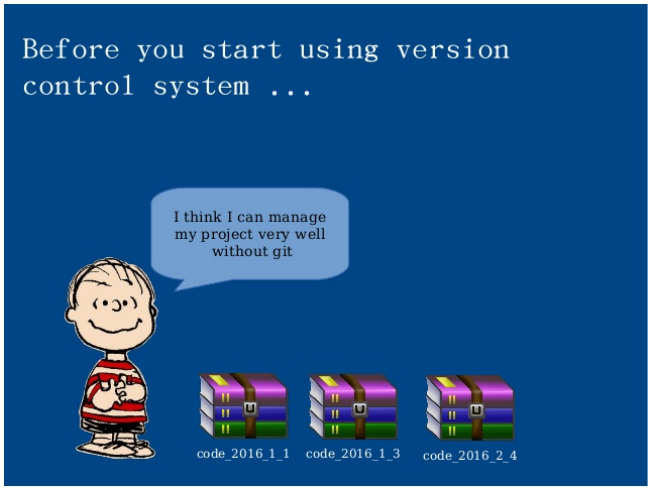
Charles VAN GOETHEM

Problématique Ici le but est d'apprendre à maîtriser la solution *git* pour le développement de Nenufaar. L'objectif principal est de comprendre comment fonctionne *git* et son utilité pour un développement collaboratif. Dans un premier temps, nous allons expliquer qu'est-ce que *git* puis nous prendrons un exemple de développement.

1 *git* : kézako ?

1.1 Comment on gère un projet ?

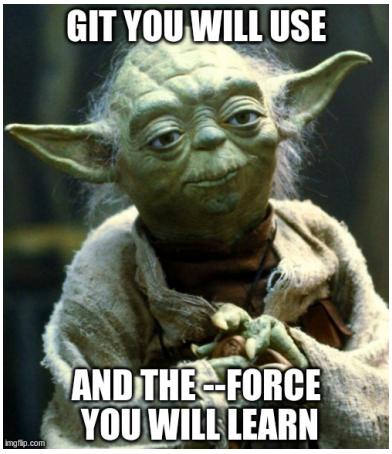
Avant d'utiliser un gestionnaire de versions, tu te dis que tu peux très bien gérer tes versions tout seul. Tu t'organises et décides de nommer tes fichiers avec leurs versions respectives. Puis tu crées des archives contenant chacune des versions de productions. Pour quoi apprendre à utiliser un logiciel pour faire ça ?



Puis tu continues de développer et tu crées de plus en plus de versions. C'est cool jusqu'à ce qu'un collègue arrive et nous demande la version *"tu sais avec la fonction qui faisait le truc là... C'était mieux !"* et que tu recherches la bonne version...

Maintenant, tu sais pourquoi on a besoin d'un gestionnaire de versions !

1.2 Les gestionnaires de versions

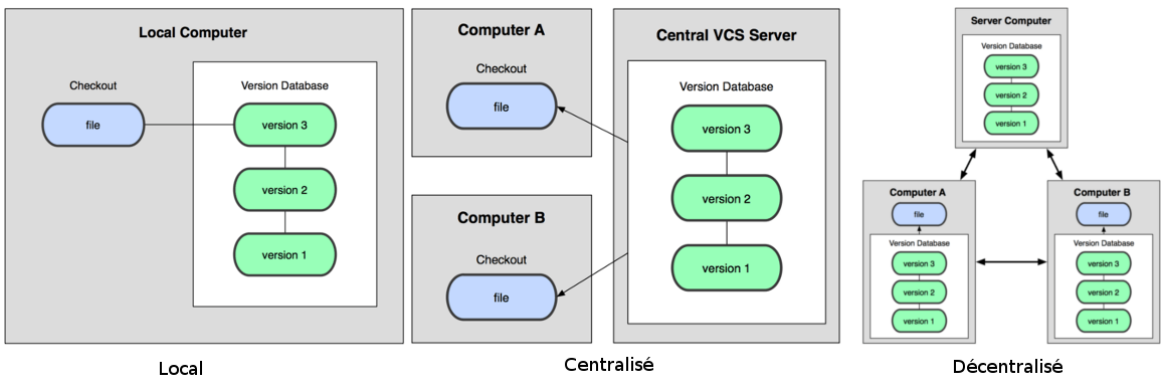


On s'organise un peu et on va voir le vieux sage du coin. Il nous explique alors qu'à son époque, on savait coder et que l'on utilisait *sccs* pour gérer les versions. Il t'explique que *sccs* signifie *Source Code Control System* et permet de gérer les versions d'un programme ! C'est ce que tu veux, mais c'est un peu vieux (1972) et puis c'est qu'en local... Le vieux sage t'explique alors qu'il existe des logiciels plus récents tels que *subversion*, développé à la fin des années 90 et reposant sur le principe de développement centralisé, ou *git*, développé au milieu des années 2000 et permettant de travailler de manière décentralisée. *git* est parmi les plus répandus aujourd'hui avec plus de 12 millions d'utilisateurs et en plus c'est libre !

"La route est longue mais la voie est libre..." — framasoft.org

Pour résumer, un gestionnaire de version est donc un outil qui va permettre d'enregistrer les évolutions d'un projet au cours du temps. Et il existe 3 systèmes différents :

- local : très bien quand on travaille seul (*sccs* – 1972)
- centralisé : chacun travaille sur la version qu'il veut (*subversion* – 1998)
- décentralisé : tout le monde peut disposer de toutes les versions (*git* – 2005)



2 La base de *git*

"Apprendre git, ce n'est pas si simple" — Boromir, fils de Denethor.

Mais en vérité utiliser *git* c'est comme tout ça s'apprend. Ce petit guide va permettre d'utiliser la base de *git* mais il restera encore beaucoup à apprendre si vous le souhaitez !

2.1 Utiliser *git*

2.1.1 Configurer *git*

git nécessite d'être configuré avant utilisation. Dans un premier temps, on doit préciser à *git* qui on est et avec quel éditeur de texte tu préfères l'utiliser :

```
$ git config --global user.name "Foo Bar"
$ git config --global user.email foo.bar@example.com
$ git config --global core.editor vim
```

Là c'est cool *git* sait qui tu es et avec quel outil tu souhaites travailler. On ne sait jamais vérifions que *git* sache bien qui on est :

```
$ git config user.name
Foo Bar
```

On peut donc vérifier une par une les infos ou alors toutes d'un coup avec la commande suivante :

```
$ git config --list
user.name=Foo Bar
user.email=foo.bar@example.com
color.diff=auto
color.status=auto
color.branch=auto
push.default=matching
core.editor=vim
```

2.1.2 Initialiser un projet

Bon, maintenant on va voir comment créer un nouveau projet. Sans *git* c'est facile, on crée un nouveau répertoire :

```
$ mkdir Unicorn_project
```

Et avec *git* pas d'affolement, c'est très simple aussi. On utilise *git* pour initialiser notre projet :

```
$ git init Unicorn_project
```

Félicitation, dans les deux cas, vous avez créé votre premier projet ! Par contre, on est en droit de se poser la question : *Pourquoi, diable faire comme cela ?*

Et bien en utilisant la commande *git init* on crée également le répertoire *.git*. Ce répertoire contient toutes les infos dont *git* a besoin pour gérer les versions.

2.1.3 Travailler

C'est bien beau tout ça, mais il faut travailler maintenant. Pour apprendre les bases de *git* on va supposer que l'on travaille seul.

Alors, imaginons que l'on crée un super code :

```
$ vi my_awesome_script.pl
$ more my_awesome_script.pl
#/usr/bin/perl

print "My little pony !\n";
```

Voilà notre premier script du projet fini ! On peut utiliser la commande suivante pour voir l'avancement du projet.

```
$ git status
Sur la branche master

Validation initiale

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

my_awesome_script.pl

aucune modification ajoutée à la validation mais des fichiers non suivis sont
présents (utilisez "git add" pour les suivre)
```

Là *git* nous signale que l'on a un fichier qui n'est pas *"suivi"*. Cela signifie que ce fichier n'est pas encore enregistré par *git*. Il faut donc préciser à *git* que l'on utilise ce fichier (on verra plus tard comment ignorer certains fichiers), et on vérifie :

```
$ git add my_awesome_script.pl
$ git status
Sur la branche master

Validation initiale

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

nouveau fichier: my_awesome_script.pl
```

Bon et maintenant il faut *"commiter"* ses actions. *"Commiter"* son code signifie soumettre, archiver son code. En même temps que l'on *commit* le code on doit ajouter un message qui définit la modification effectuée. Traditionnellement le message "Initial commit" est associé au premier *commit*.

```
$ git commit -m "Initial commit"
[master (commit racine) e023001] Initial commit
 1 file changed, 3 insertions(+)
 create mode 100644 my_awesome_script.pl
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Bon, ben déjà avec ces quelques commandes on peut archiver nos projets de façon propre. C'est pas mal pour un début !

2.1.4 Ignorer des fichiers

Bon alors, c'est déjà pas mal ! Maintenant, on va apprendre à ignorer des fichiers. En effet, dans un grand nombre de projets on génère des fichiers qu'il n'est pas utile d'archiver (notamment des fichiers temporaires, des logs...).

C'est pourquoi on peut indiquer à *git* que certains de nos fichiers ne sont pas à prendre en compte. Ici on va voir deux exemples : un répertoire contenant les logs et des fichiers *.bak* générés par un éditeur.

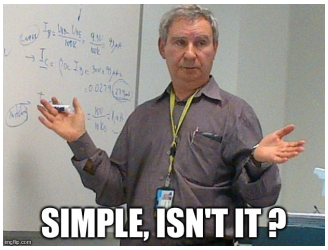
```
$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
PinkUnicornOfLove.sh
PinkUnicornOfLove.sh.bak
log/

aucune modification ajoutée à la validation mais des fichiers non suivis sont
présents (utilisez "git add" pour les suivre)
```

On a ici un script (*PinkUnicornOfLove.sh*), un fichier temporaire (*PinkUnicornOfLove.sh.bak*) et un répertoire de log. Pour ignorer ces fichiers et répertoires, on va créer un fichier particulier :

```
$ vi .gitignore
$ more .gitignore
*.bak
log/
$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
.gitignore
PinkUnicornOfLove.sh

aucune modification ajoutée à la validation mais des fichiers non suivis sont
présents (utilisez "git add" pour les suivre)
```



Si l'on compare le résultat du *git status* après création du fichier *.gitignore* avec le précédent, on s'aperçoit que les fichiers se terminant par *.bak* et le répertoire *log/* ne sont plus pris en compte dans *git*. Effectivement, ils n'apparaissent plus dans la liste des fichiers non-suivis.
Et voilà, on sait maintenant utiliser *git*. Au moins pour gérer de petits projets.

2.2 Travailler à plusieurs

Maintenant, on va compliquer un peu les choses avec le travail à plusieurs, mais en vérité c'est pas beaucoup plus complexe (en tout cas pour le début).

2.2.1 cloner un dépôt

Dans un premier temps, on va apprendre à cloner un dépôt déjà existant. Cela veut dire que l'on va créer une copie exacte du premier projet. Le clone va notamment permettre d'avoir sa propre copie du projet en local. On verra ensuite comment mettre à jour sa copie et envoyer ses propres modifications. Les deux copies auront donc le même historique et sont donc toutes les deux autant légitimes l'une que l'autre.

```
$ git clone Path/project_A Path/project_b
Clonage dans 'project_b'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 14 (delta 5), reused 0 (delta 0)
Réception d'objets: 100% (14/14), 5.09 KiB | 0 bytes/s, fait.
Résolution des deltas: 100% (5/5), fait.
Vérification de la connectivité... fait.
```

2.2.2 Maintenir à jour sa copie

On va partir de l'hypothèse que le dépôt *project_A* est la référence et que le dépôt *project_b* est la copie locale. On suppose que des modifications sont apportés au dépôt A sans que vous ayez modifié le dépôt B.

Lorsque vous ferez un *git status* dans le dépôt B, vous ne verrez pas de modification. En effet, il est à jour avec lui-même, il faut tester si le dépôt B est à jour avec le dépôt A.

```
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
rien à valider, la copie de travail est propre
$ git remote update
Récupération de origin
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 1 (delta 0), pack-reused 0
Dépaquetage des objets: 100% (3/3), fait.
Depuis Path/project_A
   69e3c98..1bb5353  master     -> origin/master
$ git status
Sur la branche master
Votre branche est en retard sur 'origin/master' de 1 commit, et peut être mise
à jour en avance rapide.
  (utilisez "git pull" pour mettre à jour votre branche locale)
rien à valider, la copie de travail est propre
```

git nous signale que nous ne sommes plus à jour par rapport au dépôt A. Il nous précise de faire un *git pull*. Cette commande permet de récupérer les modifications sur le dépôt A.

```
$ git pull
Mise à jour 69e3c98..1bb5353
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On voit que l'on a récupéré deux modification sur le fichier README.md.

2.2.3 Envoyer ses modifications

Maintenant que l'on sait maintenir à jour un dépôt, on va voir comment envoyer ses modifications. Là aussi c'est simple une commande permet de le faire :

```
$ vim README.md
$ git commit -m "little"
[master 26074e3] little
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Décompte des objets: 3, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 364 bytes | 0 bytes/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local objects.
To https://github.com/...../
   1bb5353..26074e3  master -> master
```

Et voilà la modification est envoyée ! C'est ultra-simple non ? Bon alors évidemment là on suppose que l'on a bien fait un pull juste avant de modifier notre programme et que personne ne l'a modifié au même endroit... Pour bien comprendre la suite il faut savoir que la

commande *git push* est un alias de *git push origin master* ce qui signifie que l'on envoi vers le lien d'origine (*git remote -v*) sur la branche *master*.

2.2.4 La théorie des branches

Les branches sont très importantes sous *git*. On verra dans la section suivante avec un exemple concret sur Nenufaar, comment on pourra les utiliser.

Pour faire simple les branches sont des copies d'une autre branche (la première et principale est la branche *master*). On va notamment souvent trouver une branche stable qui servira aux versions des projets (la branche de production, souvent *master*) et une seconde branche qui servira a stocker les avancer régulière entre deux points stables (la branche de développement). Mais on va également pouvoir avoir un ensemble d'autres branches qui seront des corrections de bugs par exemple.

Sur ce graphique, on distingue 3 points stables de versions (en vert) sur la branche *master*. Ensuite, on trouve une branche de développement (en orange), puis des branches concernant les features, ajout de nouvelles fonctions au programme (en bleu). La philosophie des auteurs, pour ce cas, est :

- pas de commit sur les branches *master* et *dev* ;
- une branche par feature ;
- les *features* sont fusionnées sur la branche *dev*.

Je ne vais pas aller beaucoup plus loin sur les branches puisque je l'expliquerais à l'aide d'un exemple concret plus tard. Je vais quand même expliquer les quelques commandes utiles pour leur manipulation.

Les commandes suivantes servent à créer, rejoindre, fusionner et supprimer des branches. Il faut également penser à envoyer ses modifications sur la branche spécifiée.

```
$ git branch MyBranch ### créer la branche MyBranch
$ git checkout MyBranch ### rejoindre la branche Mybranch
$ git checkout -b MyBranch ### créer et rejoindre la branche Mybranch
$ git merge MyBranch ### fusionner la branch MyBranch avec la
branche courante
$ git branch -d Mybranch ### supprimer la branche MyBranch
$ git push origin Mybranch ### envoyer sur la branche MyBranch
```

2.2.5 Résolution de conflits

Bon la théorie, c'est bien mais je tiens à vous le dire maintenant : vous aurez des erreurs ! C'est inévitable à force d'utiliser *git* on se retrouvera avec des conflits de versions. Mais avant de paniquer, on va voir ce que c'est et apprendre à les gérer.

Déjà un conflit interviendra majoritairement au moment de la fusion de deux branches (*git merge*). On pourra avoir des conflits si deux personnes ont travaillés ensemble sur le même point (même fichier, même zone).

Lors d'un conflit, on se retrouvera avec des choses du genre suivant :

```
$ git merge myBranchFromB
Auto-merging firstFile.txt
CONFLICT (content): Merge conflict in firstFile.txt
Automatic merge failed; fix conflicts and then commit the result.
$ cat firstFile.txt
This is the first line of my first file...
... and this is the second line
Now we add another line

Adding a fourth line
Adding a fifth line
modification from repoA
modifications from repoB
<<<<<<< HEAD
conflict from repoA
=======
conflict from repoB
>>>>>>> myBranchFromB
- See more at: ....
```

Pour le résoudre, il n'y a pas d'autre choix que de le faire à la main ! Eh oui *git* ne pourra pas deviner qui a raison et tort, ou si les deux sont justes/fausses ! On va donc éditer la zone entre les chevrons avec notre éditeur préféré.

3 Nenufaar

Dans cette partie, on va voir comment on peut utiliser *git* dans le cadre du développement de Nenufaar. On va configurer le compte *bitbucket* puis voir un exemple de dév. L'exemple n'est pas à effectuer, mais il permet de comprendre comment utiliser *git* dans ce contexte.

3.1 Bitbucket

Bitbucket est un service d'hébergement et de gestion de développement logiciel. En premier lieu crée votre compte sur *https://bitbucket.org*. Ensuite, envoyez moi vos identifiants pour que je puisse vous rajouter au projet. Vous recevrez un lien par mail pour vous ajouter au projet. Et voilà, vous êtes ajouté à l'équipe IURC ! Vous pouvez maintenant cloner le projet Nenufaar !

3.1.1 Configurer la connexion SSH

SSH : kézako ? SSH signifie *Secure SHell*. C'est un protocole de connexion sécurisé entre deux machines. Cela permet donc de communiquer, d'échanger des données, entre un client et un serveur par chiffrement, ici nous utilisons le chiffrement RSA (du nom de ses inventeurs Rivest, Shamir, Adleman). En effet, *Bitbucket* nous offre la possibilité de relier notre machine à notre compte à l'aide de ce protocole.

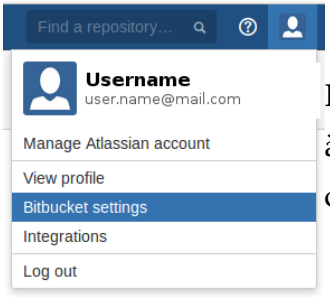
Générer la clé RSA Aller dans votre terminal préféré et lancer la commande suivante pour savoir si vous avez une clé rsa :

```
$ more ~/.ssh/id_rsa.pub
ssh-rsa sdfZTZETZGgzgze45...64564egdGDFGd9_90DSGF user@computer
```

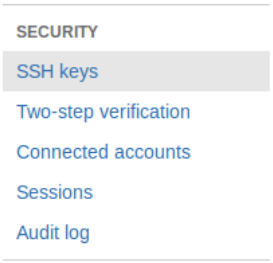
Si vous obtenez un truc du genre, vous avez une clé RSA, sinon on doit générer notre clé rsa à l'aide des commandes suivantes :

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:zPZxXxrdNDGvTEA20wHAn0FFiV34zYyFDha964gW5Ow user@computer
The key's randomart image is:
+---[RSA 2048]----+
|     .+o+BOB+o   |
|      o+o=oo=+o  |
|       o+.o+.B+..|
|      .= .+0=ooo.|
|      .ES . + o .|
|       . + o +   |
|            .    |
+----[SHA256]-----+
```

Et voilà votre clé RSA toute belle ! On la copie et on passe au niveau suivant !

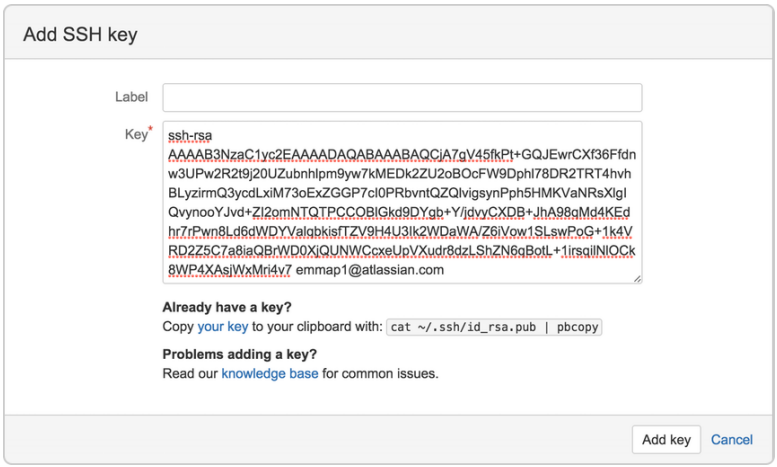


Relier son compte à sa clé Pour cela on commence par se connecter à son compte *Bitbucket*. Ensuite rendez-vous dans les paramètre du compte *avatar* -> Bitbucket settings.



Maintenant dans le menu de gauche cliquer sur *SECURITY* -> SSH keys

Vous arriverez sur un panneau qui vous propose d'ajouter une nouvelle clé. Cliquer sur *Add key*. Une fenêtre s'ouvre et vous propose d'ajouter une clé SSH. Copier le contenu du fichier *~/.ssh/id_rsa.pub*. Vous pouvez rajouter un label pour savoir quelle clé correspond à quelle machine.



3.2 Un exemple concret

C'est cool ça ! Si l'on prenait un cas concret maintenant ! On va supposer que je suis chargé d'intégrer une nouvelle fonction à Nenufaar (par exemple : intégrer un caller somatique). Chaque étape se déroulera comme suit :

- un paragraphe qui décrit ce que l'on fait ;
- les lignes de codes correspondantes ;
- un graphique de l'avancement du projet.

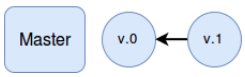
3.2.1 Le début

La première chose à faire quand on intègre un projet c'est de le cloner. Ensuite, on peut directement, ou non, avoir des tâches à effectuer. Quelques jours après avoir cloné Nenufaar, une nouvelle version est sortie. Nenufaar est passé de la version 0 à la version 1. Je dois donc mettre à jour ma copie.

```
$ git clone git@bitbucket.org:iurc/nenufaar.git
Clonage dans 'nenufaar'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 14 (delta 5), reused 0 (delta 0)
Réception d'objets: 100% (14/14), 5.09 KiB | 0 bytes/s, fait.
Résolution des deltas: 100% (5/5), fait.
Vérification de la connectivité... fait.

$ git remote update
Récupération de origin
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Dépaquetage des objets: 100% (3/3), fait.
Depuis bitbucket.org:iurc/nenufaar
   654a020..4e3b679  master     -> origin/master

$ git pull
Mise à jour 654a020..4e3b679
Fast-forward
 README.md | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```



3.2.2 Développement

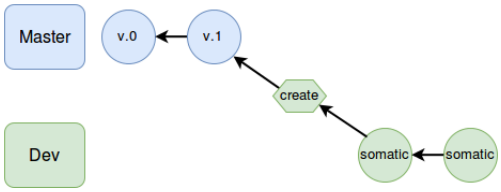
Je vais maintenant m'attaquer au développement afin d'intégrer un variant caller somatic. La première chose à faire et de créer une branche destinée à l'ajout de cette fonctionnalité. Je fais mes dév tranquilles en faisant des commit régulièrement. Et je *push* mes modifications.

```
$ git checkout -b DEV_caller_somatique
Basculement sur la nouvelle branche 'DEV_caller_somatique'

$ vim README.md

$ git commit README.md -m "add variant caller"
[DEV_caller_somatique 134853e] add variant caller
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin DEV_caller_somatiqueDécompte des objets: 3, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 349 bytes | 0 bytes/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
remote:
remote: Create pull request for DEV_caller_somatique:
remote:   https://bitbucket.org/iurc/nenufaar/pull-requests
/new?source=DEV_caller_somatique&t=1
remote:
To git@bitbucket.org:iurc/nenufaar.git
 * [new branch]      DEV_caller_somatique -> DEV_caller_somatique
```



3.2.3 Bugfix

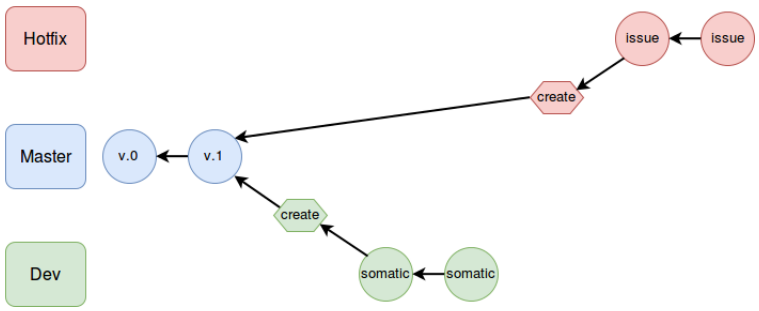
Je reçois un coup de téléphone de David, parti en conférence à New-York. Il m'explique alors qu'un utilisateur de Nenufaar a remarqué un bug (ou bogue en français). Il faut le corriger en urgence et il n'a pas le temps de le faire (jetlag, conférence, champagne tout ça quoi...), je dois donc le corriger. Je commence par créer une branche portant un numéro de bug par exemple (bugfix29). Et je m'attelle à la correction du Bug.

```
$ git checkout master
$ git checkout -b bugfix29
Basculement sur la nouvelle branche 'bugfix29'

$ vim README.md

$ git commit README.md -m "fix bug"
[bugfix29 fe98406] fix bug
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin bugfix29
Décompte des objets: 3, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 335 bytes | 0 bytes/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
remote:
remote: Create pull request for bugfix29:
remote:   https://bitbucket.org/iurc/nenufaar/pull-requests
/new?source=bugfix29&t=1
remote:
To git@bitbucket.org:iurc/nenufaar.git
 * [new branch]      bugfix29 -> bugfix29
```



3.2.4 Relier au master

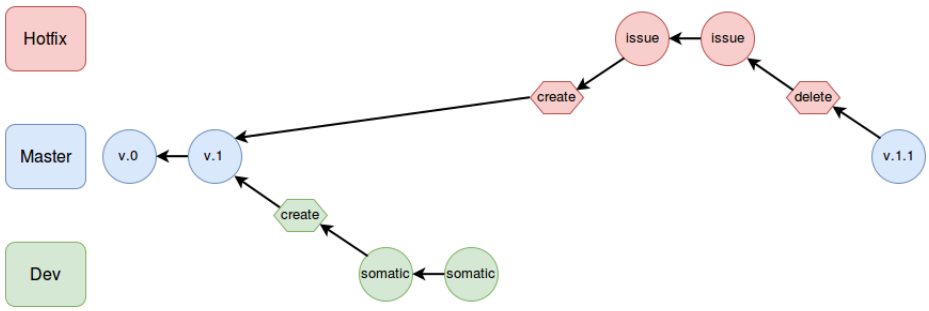
Après avoir corrigé le bug, je peux fusionner la branche contenant le *bugfix* avec la branche master. Puis supprimer la branche correspondant à la suppression du bug. Enfin, j'envoie les modifications sur le serveur.

```
$ git checkout master
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git merge bugfix29
Mise à jour 4e3b679..fe98406
Fast-forward
 README.md | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git branch -d bugfix29
Branche bufix29 supprimée (précédemment fe98406).

$ git push
Total 0 (delta 0), reused 0 (delta 0)
To git@bitbucket.org:iurc/nenufaar.git
   4e3b679..fe98406  master -> master
```

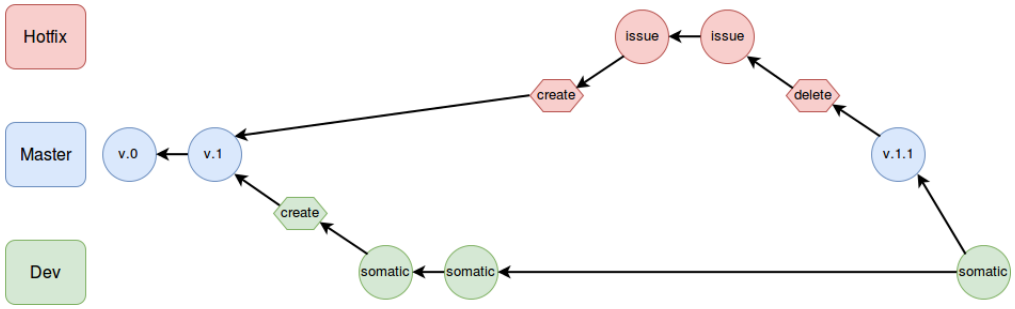


3.2.5 Mise à jour de la *feature*

Je me replace maintenant au niveau de la branche de développement de la prochaine *feature* sur laquelle je travaille actuellement. Il suffit que je fusionne la branche de développement avec la branche *master* pour avoir la correction du bug.

```
$ git checkout DEV_caller_somatique
Basculement sur la branche 'DEV_caller_somatique'

$ git merge master
Fusion automatique de README.md
Merge made by the 'recursive' strategy.
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```



3.2.6 Fin de développement de la *feature*

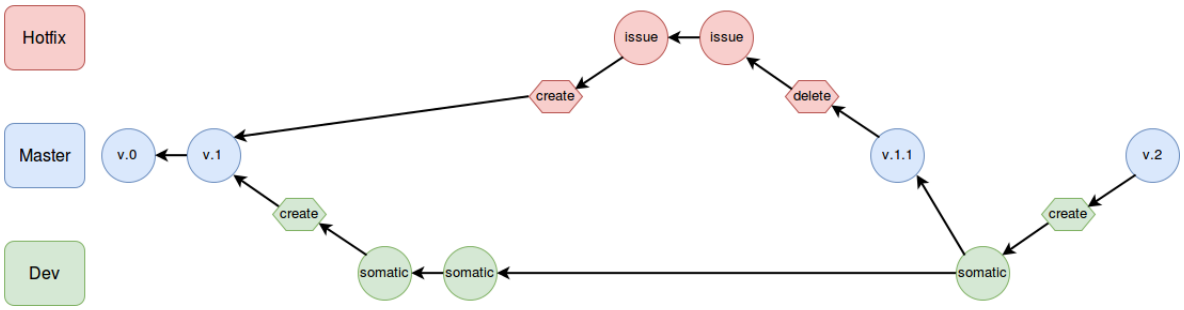
Après avoir finit de développer ma *feature* je peux l'intégrer dans la branche *master*. J'envoie mes modifications et c'est fini !

```
$ git checkout master
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git merge DEV_caller_somatique
Mise à jour fe98406..3ea1853
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

 $ git branch -d DEV_caller_somatique
Branche DEV_caller_somatique supprimée (précédemment 3ea1853).

$ git push
Décompte des objets: 3, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 398 bytes | 0 bytes/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
To git@bitbucket.org:iurc/nenufaar.git
   fe98406..3ea1853  master -> master
```



4 Un peu plus loin

Voici quelques liens utiles pour aller plus loin et qui m'ont servi à rédiger ce petit cours :

— bioinfo-fr.net/git-premiers-pas
— git-scm.com/doc
— www.slideshare.net/shengwen1997/brief-tutorial-on-git
— rogerdudler.github.io/git-guide/index.fr.html
— www.git-tower.com
— formation-debian.via.ecp.fr/ssh.html
— doc.ubuntu-fr.org/ssh

fournova

GIT CHEAT SHEET

presented by **TOWER** › Version control with Git - made easy

CREATE

Clone an existing repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository

```
$ git init
```

LOCAL CHANGES

Changed files in your working directory

```
$ git status
```

Changes to tracked files

```
$ git diff
```

Add all current changes to the next commit

```
$ git add .
```

Add some changes in <file> to the next commit

```
$ git add -p <file>
```

Commit all local changes in tracked files

```
$ git commit -a
```

Commit previously staged changes

```
$ git commit
```

Change the last commit

Don‘t amend published commits!

```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest

```
$ git log
```

Show changes over time for a specific file

```
$ git log -p <file>
```

Who changed what and when in <file>

```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches

```
$ git branch -av
```

Switch HEAD branch

```
$ git checkout <branch>
```

Create a new branch based on your current HEAD

```
$ git branch <new-branch>
```

Create a new tracking branch based on a remote branch

```
$ git checkout --track <remote/branch>
```

Delete a local branch

```
$ git branch -d <branch>
```

Mark the current commit with a tag

```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes

```
$ git remote -v
```

Show information about a remote

```
$ git remote show <remote>
```

Add new remote repository, named <remote>

```
$ git remote add <shortname> <url>
```

Download all changes from <remote>, but don‘t integrate into HEAD

```
$ git fetch <remote>
```

Download changes and directly merge/integrate into HEAD

```
$ git pull <remote> <branch>
```

Publish local changes on a remote

```
$ git push <remote> <branch>
```

Delete a branch on the remote

```
$ git branch -dr <remote/branch>
```

Publish your tags

```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD

```
$ git merge <branch>
```

Rebase your current HEAD onto <branch>

Don‘t rebase published commits!

```
$ git rebase <branch>
```

Abort a rebase

```
$ git rebase --abort
```

Continue a rebase after resolving conflicts

```
$ git rebase --continue
```

Use your configured merge tool to solve conflicts

```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved

```
$ git add <resolved-file>
$ git rm <resolved-file>
```

UNDO

Discard all local changes in your working directory

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

Discard local changes in a specific file

```
$ git checkout HEAD <file>
```

Revert a commit (by producing a new commit with contrary changes)

```
$ git revert <commit>
```

Reset your HEAD pointer to a previous commit

...and discard all changes since then

```
$ git reset --hard <commit>
```

...and preserve all changes as unstaged changes

```
$ git reset <commit>
```

...and preserve uncommitted local changes

```
$ git reset --keep <commit>
```