Table des matières

1 Préambule

Git permet de gérer un projet (de programmation généralement) et de garder en mémoire l'historique de toutes les versions d'un ensemble de fichiers. Il permet de gérer un projet à plusieurs, de programmer afin de pouvoir revenir en arrière, comparer avec d'anciennes versions et cie.

Le principe est d'avoir un serveur git (un seul possible) qui va garder en mémoire l'historique de toutes les versions et un client git (plusieurs possibles) qui vont se connecter au serveur pour mettre à jour la version des fichiers ou en récupérer les dernières versions.

Remarque : Il est possible que le serveur soit lui aussi client, dans le cas où il n'y aurait qu'un seul développeur et qu'on ne souhaite pas passer par internet.

1.1 Configuration

Afin d'avoir la coloration syntaxique, il faut faire :

```
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

De même, il faut configurer votre nom (ou pseudo) :

```
git config --global user.name "votre_pseudo"
```

Puis votre e-mail :

```
git config --global user.email moi@email.com
```

Vous pouvez aussi éditer votre fichier de configuration .gitconfig situé dans votre répertoire personnel pour y ajouter une section alias à la fin :

```
vim ~/.gitconfig
```

```
[color]
        diff = auto
        status = auto
        branch = auto
[user]
        name = votre_pseudo
        email = moi@email.com
[alias]
        ci = commit
        co = checkout
        st = status
        br = branch
```

1.2 Git sur internet

Je vais prendre l'exemple de google code, qui est celui que j'ai choisi et que je suis en train d'apprendre.

Une fois le projet créé (sur la page http://code.google.com/hosting/createProject), il faut faire la commande suivante pour récupérer le contenu du projet localement (et pouvoir envoyer les modifs par la suite) :

```
cd Formulaire
git clone https://autiwa@code.google.com/p/autiwa-tutorials/
```

Cette commande permet de récupérer le contenu du projet et de le copier dans un dossier **Formulaires** qui sera créé dans le dossier courant.

Définition 1 (Clone)
Opération d'extraction d'une version d'un projet du repository vers un répertoire de travail local.

1.3 Git localement

Le serveur ET le client seront alors sur la même machine. Pour créer un projet, il faut créer un dossier, par exemple dans mon cas un dossier **mercury** dans mon **$HOME**, puis je fais, dans mon répertoire utilisateur :

```
cd mercury
git init
```

pour un projet que j'appelle **mercury**.

2 Commandes universelles git

Ici, je note les commandes qui sont valables à la fois pour svn installé sur un serveur internet, ou sur une machine locale pour un usage personnel

2.1 Mettre à jour sa copie locale : pull

```
git pull
```

Pour celà, il faut que le dossier dans lequel on se trouve ait déjà été défini comme un dossier git via un *clone* (voir [§ 1.3])

2.2 Lister les modifications locales : status

La commande git status vous indique les fichiers que vous avez modifiés récemment :

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

2.3 Ajouter des fichiers ou dossiers au projet : add

Pour ajouter des fichiers il faut faire :

```
git add latex/ vim/
```

où **latex**/ et **vim**/ sont deux dossiers existant dans le dossier local de référence

> **Remarque** : Au cas où ça serait pas clair. J'ai créé un dossier /**home**/**autiwa**/**Formulaires** grâce à [§ 1.3]. Dans ce dossier, j'ai créé et rempli à la main les sous-dossiers **latex**/ et **vim**/. Maintenant, grâce à la commande ci-dessus, je définis ces sous-dossiers comme étant rattachés au projet. En faisant ainsi le contenu est rajouté récursivement.

⚠ Cette commande n'agit que sur le répertoire local (la *working copy*). Il faut ensuite appliquer ces changements au dépot (voir [§ 2.5 on the following page]) pour les valider.

Supposons que vous veniez d'ajouter un fichier à Git avec git add et que vous vous apprêtiez à le « commiter ». Cependant, vous vous rendez compte que ce fichier est une mauvaise idée et vous voudriez annuler votre git add.

Il est possible de retirer un fichier qui avait été ajouté pour être « commité » en procédant comme suit :

```
git reset HEAD -- fichier_a_supprimer
```

2.4 Voir les différences versions : diff

```
git diff
```

Remarque : Il est possible de regarder les différences sur un fichier en particulier.

2.5 Enregistrer les modifications localement : commit

Pour mettre à jour les versions sur serveur à partir des modifications effectuées localement, il faut :

```
git commit -a -m "initialisation"
```

où **"initialisation"** est le commentaire qui décrit la mise à jour et les modifications effectuées.

 Un commit avec git est local : à moins d'envoyer ce commit sur le serveur comme on apprendra à le faire plus loin, personne ne sait que vous avez fait ce commit pour le moment. Cela a un avantage : si vous vous rendez compte que vous avez fait une erreur dans votre dernier commit, vous avez la possibilité de l'annuler (ce qui n'est pas le cas avec SVN !).

2.6 Propager les modifications locales sur le dépots du serveur : push

On peut modifier ces commits là, tant qu'ils sont en local. Puis une fois vérifié. Il ne reste plus qu'à les envoyer sur le serveur. Pour celà on vérifie tout d'abord ce qu'on s'apprête à envoyer, et on envoie sur le serveur :

```
git log -p
git push
```

2.7 Annuler les modifications non commités : checkout

Pour annuler les modifications locales et revenir à la dernière révision :

```
git checkout *.f90
```

va remettre en l'état tous les fichiers du dépot qui sont présents dans le dossier courant (on peut aussi ne préciser qu'un seul fichier).

2.8 Annuler un commit effectué par erreur

2.8.1 Modifier le message du dernier commit

```
git commit --amend
```

2.8.2 Annuler un commit sans toucher aux fichiers (soft)

```
git reset HEAD^
```

annule le dernier commit et revient à l'avant dernier. Pour autant les fichiers ne sont pas modifiés, seul le commit en lui même est annulé. Pour annuler les modifications liées au commit, il faut faire un hard reset (section suivante).

2.8.3 Annuler un commit et les modifications de fichier associées (hard)

```
git reset --hard HEAD^
```

Annule le dernier commit et toutes les modifications qui s'y rapportent.

Pour indiquer à quel commit on souhaite revenir, il existe plusieurs notations :

- `HEAD` : dernier commit ;
- `HEAD^` : avant-dernier commit ;
- `HEAD^^` : avant-avant-dernier commit ;
- `HEAD~2` : avant-avant-dernier commit (notation équivalente) ;
- `d6d98923868578a7f38dea79833b56d0326fcba1` : indique un numéro de commit précis ;
- `d6d9892` : indique un numéro de commit précis (notation équivalente à la précédente, bien souvent écrire les premiers chiffres est suffisant tant qu'aucun autre commit ne commence par les mêmes chiffres).

2.8.4 Annuler un commit publié

Si vous publiez un commit sur le serveur, mais que vous souhaitez l'annuler, c'est quand même possible, mais c'est un peu plus barbare qu'un commit non encore propagé (à l'aide de push).

Pour celà il faut créer un nouveau commit qui annule les modifications du commit que vous souhaitez annuler. Il faut alors connaître l'ID du commit visé et faire :

```
git revert 6261cc2
```

Il faut préciser l'ID du commit à « revert ». Il n'est pas obligatoire d'indiquer l'ID en entier (qui est un peu long), il suffit de mettre les premiers chiffres tant qu'ils sont uniques (les 4-5 premiers chiffres devraient suffire). On vous invite à entrer un message de commit. Un message par défaut est indiqué dans l'éditeur.

Une fois que c'est enregistré, le commit d'annulation est créé. Il ne vous reste plus qu'à vérifier que tout est bon et à le publier (avec un git push).

2.9 Historique des modifications : log

La commande `log` permet de voir un historique des commentaires de révision, soit de l'ensemble des fichiers, soit d'un fichier en particulier.

```
git log
```

affiche l'historique de toutes les révisions.


Chaque commit est numéroté grâce à un long numéro hexadécimal comme 12328a1bcbf231da-8eaf942f8d68c7dc0c7c4f38. Cela permet de les identifier.

3 Travailler avec des branches

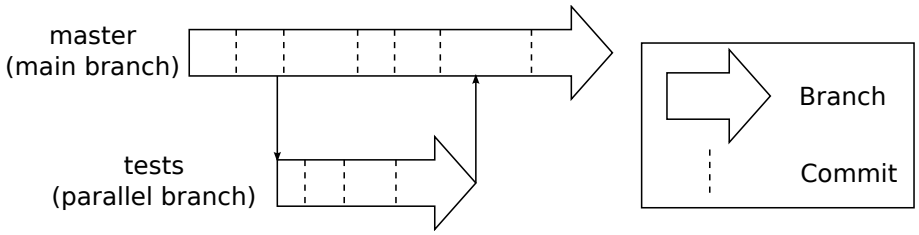

FIGURE 1 – Le principe des branches est de travailler à part sur des modifications et de ne les fusionner avec la branche principale que si c'est ok et ça donne quelque chose.

L'idée derrière le fait de créer des branches, c'est de bien séparer les idées de développement, et de pouvoir les tester et les séparer avant de les inclure (ou pas) dans la branche principale.

> En gros, Si vous avez une modification à faire, qui risque de prendre un peu de temps et qui va nécessiter plusieurs commit, faites une branche !

3.1 Lister les branches

Première chose utile, lister les branches existantes. Par défaut il n'y aura que la branche principale :

```
git branch
```

3.2 Créer une branche

Pour créer une branche **test_01**, il faut utiliser la commande :

```
git branch test_01
```

Si vous listez alors les branches, vous aurez :

```
$ git branch
* master
  test_01
```

où on voit que **master** est toujours la branche active. Il faut donc changer de branche pour pouvoir modifier les fichiers et faire des commits dans cette branche.

3.3 Activer une branche et y faire des commits

Il faut déclarer cette branche comme active via :

```
git checkout test_01
```

La liste des branches donne alors :

```
$ git branch
  master
* test_01
```

qui montre que la branche **test_01** est maintenant active.

Remarque : **git checkout** est utilisé pour changer de branche mais aussi pour restaurer un fichier tel qu'il était lors du dernier commit. La commande a donc un double usage.

⚠ Vous pouvez changer d'une branche à l'autre, faire des commits (éventuellement des push) sur la branche principale, mais ceux ci ne seront pas propagés aux autres branches, c'est d'ailleurs le principe des branches.

3.4 Fusionner une branche avec la branche principale

Pour celà, il faut se placer dans la branche principale et demander d'intégrer les changements d'une autre branche en faisant :

```
git checkout master
git merge test_01
```

Une fois fusionné, vous pouvez supprimer la branche **test_01** (voir la section suivante).

3.5 Incorporer les nouvelles modifications de la branche principale dans une branche annexe : rebase

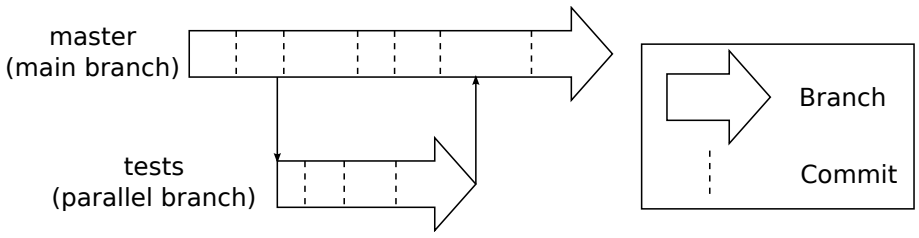

FIGURE 2 – Avec git rebase, on peut inclure les dernières modifications d'une branche dans une autre branche, afin de décaler la « base » de cette branche. En résumé, c'est comme si on faisait bifurquer la branche concernée à la fin de la branche de laquelle on veut inclure les modifications

```
git rebase master devel
```

permet d'inclure dans **devel** les dernières modifications de **master**.

En cas de conflits, on peut modifier les fichiers affectés et continuer via la commande

```
git rebase --continue
```

ou annuler directement avec la commande

```
git rebase --abort
```

3.6 Mettre de coté et reprendre un travail en cours : stash

Si on est en train de faire une modification, sans faire de commit, mais qu'on doit faire des modifications plus urgentes, et surtout immédiate, on peut souhaiter ne pas perdre le travail déjà effectué. Au lieu de faire un commit on peut simplement faire une sauvegarde de l'état modifié, pour le restituer plus tard.

```
git stash save "ongoing work about something"
```

va sauvegarder les modifications effectuées depuis le dernier commit dans le **stash**, et remettre les fichiers dans leur état lors du dernier commit.

On peut ensuite faire les modifications urgente, dans cette branche ou dans une autre, faire des commit etc. Puis une fois qu'on veut réutiliser le stash, il suffit de faire :

```
git stash apply
```

3.7 Supprimer une branche

Pour supprimer une branche dont vous avez déjà propagé les modifications dans la branche principale, faites :

```
git branch -d test_01
```

Si jamais vous souhaitez supprimer une branche sans avoir propagé les changements (si cet essai était une erreur par exemple) vous pouvez utiliser la commande suivante qui force la suppression :

```
git branch -D test_01
```

4 Avancé

4.1 Tagger une version : tag

Il est possible d'associer un tag (un nom de code) à un commit particulier, afin de le mettre en valeur, pour lui donner un numéro de version par exemple, pour les versions stables que les gens devraient utiliser.

Pour ajouter un tag sur un commit :

```
git tag NOMTAG IDCOMMIT
```

ou avec un message explicatif :

```
git tag NOMTAG IDCOMMIT -m "message associe au tag"
```

Donc dans le cas présent, on écrirait :

```
git tag v1.3 2f7c8b3428aca535fdc970feeb4c09efa33d809e
```

Un tag n'est pas envoyé lors d'un push, il faut préciser l'option –**tags** pour que ce soit le cas :

```
git push --tags
```

Maintenant, tout le monde peut référencer ce commit par ce nom plutôt que par son numéro de révision.

Pour supprimer un tag créé :

```
git tag -d NOMTAG
```

Pour lister les tags existants :

```
git tag -l
```

On peut aussi lister les tags correspondant à un motif particulier :

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Pour avoir les infos sur le tag **nomdutag** :

```
git show nomdutag
```

4.2 Ignorer des fichiers (.gitignore)

Pour ignorer un fichier dans git, créez un fichier **.gitignore** (à la racine) et indiquez-y le nom du fichier. Entrez un nom de fichier par ligne, comme ceci :

```
project.xml
dossier/temp.txt
*.tmp
cache/*
```

Aucun de ces fichiers n'apparaîtra dans git status, même s'il est modifié. Il ne paraîtra donc pas dans les commits. Utilisez cela sur les fichiers temporaires par exemple, qui n'ont aucune raison d'apparaître dans Git.

Il est possible d'utiliser une étoile (*) comme joker. Dans l'exemple précédent, tous les fichiers ayant l'extension **.tmp** seront ignorés, de même que tous les fichiers du dossier cache.

4.3 Rechercher dans les fichiers du dépot : git grep

On peut vouloir chercher quelque chose parmis les fichiers du dépot. Une commande très simple pour faire ça :

```
git grep "xmax"
```

qui va donner toutes les occurences par fichier. Mais si on veut aussi les numéros de lignes, il suffit alors de faire :

```
git grep -n "xmax"
```

5 FAQ

5.1 git demande le mot de passe à chaque fois

Il est possible de rentrer les mots de passe dans un fichier .netrc dans le répertoire utilisateur. Le fichier ressemble à ceci :

```
machine code.google.com login usernamea@gmail.com password blablabla
```

Mais si lors de la création du dépot, l'url utilisée contenait l'identifiant, git demandera le mot de passe à chaque fois. Afin de résoudre le problème, il faut se placer dans le répertoire du dépot. Puis aller dans le sous-dossier **.git** et modifier dans le fichier **config** pour enlever la partie **login@** qu'il pourrait y avoir.

6 Gérer un conflit

Index