Utilisation de Git
UCBL

Matthieu Moy

Janvier 2018

Ce document peut être téléchargé depuis l'adresse suivante :
`http://perso.univ-lyon1.fr/fabien.rico/site/projet:2018:pri:start`

1 Introduction

1.1 Git et la gestion de versions

Git est un gestionnaire de versions, c'est à dire un logiciel qui permet de conserver l'historique des fichiers sources d'un projet, et d'utiliser cet historique pour fusionner automatiquement plusieurs révisions (ou « versions »). Chaque membre de l'équipe travaille sur sa version du projet, et peut envoyer les versions suffisamment stables à ses coéquipiers via un dépôt partagé (commande `git push`) qui pourront les récupérer et les intégrer aux leurs quand ils le souhaitent (commande `git pull`).

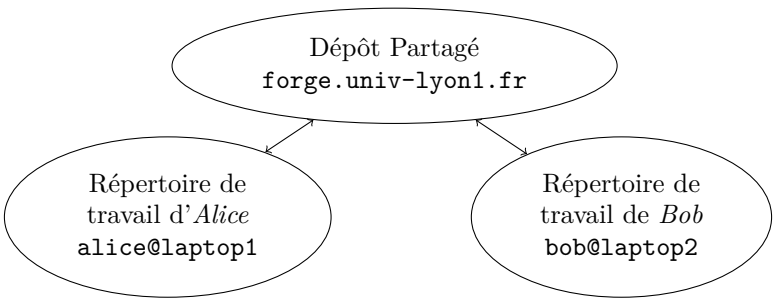
Il existe beaucoup d'autres gestionnaires de versions, par exemple Subversion (`svn`) et Mercurial (`hg`). Git est aujourd'hui le plus utilisé.

1.2 Organisation pendant la séance machine

Pour la séance machine, choisissez deux PC adjacents par équipe (on peut utiliser son ordinateur portable à la place d'un PC fixe). Chaque étudiant travaille sur son compte.

Le dépôt partagé est un répertoire qui contiendra l'ensemble de l'historique du projet, et qui restera accessible en permanence à tout le monde. Il sera hébergé sur un serveur. Ce document propose d'utiliser la forge Lyon 1 (`https://forge.univ-lyon1.fr/`), mais en adaptant les consignes sur la création du dépôt, on peut suivre ce document avec n'importe quel autre hébergeur. Vous pouvez aussi en théorie utiliser n'importe quelle machine avec Git et SSH installé pour héberger votre dépôt partagé, mais l'utilisation d'une forge facilite les choses. Le dépôt partagé sera hébergé sur le compte d'un étudiant de l'équipe sur la forge. Tous les membres de l'équipe auront accès à ce dépôt en lecture et en écriture, donc le choix du compte hébergeant le dépôt n'a pas beaucoup d'importance.

Dans la suite des explications, on suppose que l'utilisateur `alice` héberge le dépôt sur la machine `forge.univ-lyon1.fr`. L'équipe est constituée d'*Alice* (qui travaille plutôt sur son portable, `laptop1`) et *Bob*, qui travaille également sur son portable `laptop2`). Si *Alice* ou *Bob* travaille sur un PC de l'école, en remplaçant `laptop1` ou `laptop2` par le nom de la machine. Les explications sont écrites pour 2 utilisateurs pour simplifier, mais il peut y avoir un nombre quelconque de coéquipiers.



2 Configuration de Git

Si vous travaillez sur votre machine personnelle, vérifiez que Git est installé (la commande `git`, sans argument, doit vous donner un message d'aide). Si ce n'est pas le cas, installez-le (sous Ubuntu, « `apt-get install git gitk` » ou « `apt-get install git-core gitk` » devrait faire l'affaire, ou bien rendez-vous sur `http://git-scm.com/`).

On commence par configurer l'outil Git. Sur la machine sur laquelle on souhaite travailler (donc sur vos portables dans notre exemple), chaque étudiant fait :

```
git config --edit --global
```

Ou bien :

```
emacs ~/.gitconfig          # ou son éditeur préféré à la place d'Emacs !
```

Le contenu du fichier `.gitconfig` (à créer s'il n'existe pas) doit ressembler à ceci :

```
[core]
        editor = votre_editeur_prefere
[user]
        name = Prénom Nom
        email = Prenom.Nom@etu.univ-lyon1.fr
[diff]
        renames = true
# Sections ci-dessous pas nécessaires avec un Git récent
[push]
        default = simple
[color]
        ui = auto
```

La section `[user]` est obligatoire, elle donne les informations qui seront enregistrées par Git lors d'un `commit`. Il est conseillé d'utiliser votre vrai nom (pas juste votre login) et votre adresse officielle Lyon 1 ici, et d'utiliser la même configuration sur toutes les machines sur lesquelles vous travaillez (recopiez simplement le fichier `~/.gitconfig` sur toutes les machines que vous utilisez).

La ligne `editor` de la section `[core]` définit votre éditeur de texte préféré (par exemple, `emacs`, `vim`, ...). Si vous utilisez gvim, écrivez ici `gvim -f` et si vous utilisez gedit, `gedit -s` [1]. Cette dernière ligne n'est pas obligatoires ; si elle n'est pas présente, la variable d'environnement `VISUAL` sera utilisée ; si cette dernière n'existe pas, ce sera la variable d'environnement `EDITOR`.

la section `[diff]` et la section `[color]` sont là pour rendre l'interface de Git plus jolie. La section `[push]` permet d'avoir le même comportement avec Git 2.x et Git 1.x et évite un warning gênant avec Git antérieur à 2.8.

3 Messages en anglais

Git s'adapte à votre environnement et vous parlera donc français sur une machine configurée en français (c'est le cas au Nautibus). En pratique, les traductions françaises des messages sont souvent peu compréhensibles. Il est vivement recommandé de passer l'interface en anglais. Pour le faire pour l'ensemble des commandes, lancez :

```
export LANGUAGE=en_US.UTF-8
```

Pour passer uniquement la commande `git` en anglais sans modifier le reste des commandes, faites :

```
alias git='LANGUAGE=en_US.UTF-8 git'
```

Dans les deux cas, ces commandes sont locales à un shell, il faut les ajouter au fichier `~/.bashrc` pour que leur effet soit définitif.

1. Si un `gvim` est déjà lancé, la commande `git commit` va se connecter au `gvim` déjà lancé pour lui demander d'ouvrir le fichier, et le processus lancé par `git` va terminer immédiatement. Git va croire que le message de commit est vide, et abandonner le commit. Utiliser `gvim -f` permet d'ouvrir une nouvelle instance de gvim ce qui règle le problème.

4 Mise en place

Le contenu de cette section est réalisé une bonne fois pour toute, au début du projet. Si certains membres de l'équipe ne comprennent pas les détails, ce n'est pas très grave, nous verrons ce que tout le monde doit savoir dans la section 5.

4.1 Création du dépôt partagé

On va maintenant créer le dépôt partagé. Seule *Alice* fait cette manipulation, sur son compte `forge.univ-lyon1.fr` :

- Ouvrir dans un navigateur la page `http://forge.univ-lyon1.fr/`
- S'identifier avec ses identifiants Lyon 1.
- Cliquer sur « new project » en haut à droite
- Choisir un nom de projet (nous utiliserons `lifprojet` dans la suite, mais vous pouvez choisir le nom que vous souhaitez).
- Choisir si votre projet doit être public ou privé. Si vous souhaitez rendre votre projet public, demandez à votre enseignant si cela pose problème (fraude potentielle entre équipes ...).
- Validez.

Vous devriez maintenant voir la page d'accueil de votre projet, vide. L'URL de la page doit ressembler à `https://forge.univ-lyon1.fr/alice/lifprojet`.

Chaque membre de l'équipe doit s'être identifié une fois sur la forge, pour que son compte soit correctement initialisé. Mais les autres membres n'ont pas à créer le projet.

On va maintenant donner la permission aux coéquipiers :

- Choisir « settings » → « members ».
- Dans l'onglet « add members », choisir les membres de l'équipe, et leur donner le rôle « developer » ou « master ».

Chaque coéquipier peut vérifier qu'il voit bien le projet sur sa page personnelle de la forge.

4.2 Création des répertoires de travail

On va maintenant créer le premier répertoire de travail. Pour l'instant, il n'y a aucun fichier dans notre dépôt, donc la première chose à faire sera d'y ajouter les fichiers sur lesquels on veut travailler. Dans notre exemple, c'est *Alice* qui va s'en occuper.

Pour créer un répertoire de travail dans le répertoire `~/lifprojet` (qui n'existe pas encore), *Alice* entre donc les commandes :

```
cd
git clone https://forge.univ-lyon1.fr/alice/lifprojet.git
```

L'URL est obtenue depuis la page du projet : choisir HTTPS dans le menu déroulant (qui propose aussi SSH), puis copier-coller l'URL proposée.

Il est probable que vous ayez des problèmes de certificat non-reconnu (erreur ressemblant à « server certificate verification failed »). Si c'est le cas, suivez les instructions disponibles ici : `https://forge.univ-lyon1.fr/EMMANUEL.COQUERY/forge/wikis/FAQ`

Une fois ceci fait, vous aurez un répertoire `~/lifprojet` dans lequel travailler. Pour l'instant, ce répertoire est vide, ou presque : il contient un répertoire caché `.git/` qui contient les méta-données utiles à Git (c'est là que sera stocké l'historique du projet). Si on est curieux, on peut regarder le contenu du répertoire `.git` : c'est un ensemble de fichiers que Git utilise pour représenter l'état et l'historique de notre projet (les fichiers sur lesquels on travaille n'y sont pas).

Pour cette séance machine, un répertoire `sandbox/` a été prévu pour vous, pour pouvoir vous entraîner sans casser un vrai projet. *Alice* télécharge le répertoire depuis `http://perso.univ-lyon1.fr/fabien.rico/site/_media/projet:2018:pri:sandbox.tar.gz` (lien sur la page du cours), puis importe ce répertoire :

```
cd ~/lifprojet/
tar xzvf ~/chemin/vers/le/repertoire/sandbox.tar.gz
git add sandbox/
git commit -a -m "import du repertoire sandbox/"
```

La commande « `git add sandbox/` » dit à Git de « traquer » tous les fichiers du répertoire `sandbox/`, c'est à dire qu'il va enregistrer le contenu de ces fichiers, et suivre leur historique ensuite. La commande `git commit` enregistre effectivement le contenu de ces fichiers.

Alice peut maintenant envoyer le squelette qui vient d'être importé vers le dépôt partagé :

```
git push
```

Tout est prêt pour commencer à travailler. *Bob* peut à son tour récupérer sa copie de travail :

```
cd
git clone https://forge.univ-lyon1.fr/alice/lifprojet.git
cd lifprojet
ls
```

Attention, *Bob* utilise bien son nom d'utilisateur `forge.univ-lyon1.fr` pour se connecter. Vu que *Bob* ne connaît pas le mot de passe d'*Alice*, son login `bob` sur la forge est le seul moyen pour lui de se connecter à cette machine.

Si tout s'est bien passé, la commande `ls` ci-dessus devrait faire apparaître le répertoire `sandbox/`.

5 Utilisation de Git pour le développement

Pour commencer, on va travailler dans le répertoire `sandbox`, qui contient deux fichiers pour s'entraîner :

```
cd sandbox
emacs hello.c
```

Il y a deux problèmes avec `hello.c` (identifiés par des commentaires). *Alice* résout l'un des problème, et *Bob* choisit l'autre. Par ailleurs, chacun ajoute son nom en haut du fichier, et enregistre le résultat.

5.1 Création de nouvelles révision

```
git status                  # comparaison du répertoire de
                            # travail et du dépôt.
```

On voit apparaître :

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   hello.c
```

Ce qui nous intéresse ici est la ligne « modified : hello.c » (la distinction entre « Changes not staged for commit » et « Changes to be committed » n'est pas importante pour l'instant), qui signifie que vous avez modifié `hello.c`, et que ces modifications n'ont pas été enregistrées dans le dépôt. On peut vérifier plus précisément ce qu'on vient de faire :

```
git diff HEAD
```

Comme *Alice* et *Bob* ont fait des modifications différentes, le diff affiché sera différent, mais ressemblera dans les deux cas à :

```
diff --git a/sandbox/hello.c b/sandbox/hello.c
index a47665a..7f67d33 100644
--- a/sandbox/hello.c
+++ b/sandbox/hello.c
@@ -1,5 +1,5 @@
 /* Chacun ajoute son nom ici */
-/* Auteurs : ... et ... */
```

```
+/* Auteurs : Alice et ... */

 #include <stdio.h>
```

Les lignes commençant par '-' correspondent à ce qui a été enlevé, et les lignes commençant par '+' à ce qui a été ajouté par rapport au précédent commit. Si vous avez suivi les consignes ci-dessus à propos du fichier `.gitconfig`, vous devriez avoir les lignes supprimées en rouge et les ajoutées en vert.

Maintenant, *Alice* et *Bob* font :

```
git commit -a      # Enregistrement de l'état courant de
                   # l'arbre de travail dans le dépôt local.
```

L'éditeur est lancé et demande d'entrer un message de 'log'. Ajouter des lignes et d'autres renseignements sur les modifications apportées à hello.c (on voit en bas la liste des fichiers modifiés). Un bon message de log commence par une ligne décrivant rapidement le changement, suivi d'une ligne vide, suivi d'un court texte expliquant pourquoi la modification est bonne [2].

On voit ensuite apparaître :

```
[master 2483c22] Ajout de mon nom
 1 files changed, 2 insertions(+), 12 deletions(-)
```

Ceci signifie qu'un nouveau « commit » (qu'on appelle aussi parfois « revision » ou « version ») du projet a été enregistrée dans le dépôt. Ce commit est identifié par une chaîne hexadécimale (« 2483c22 » dans notre cas).

On peut visualiser ce qui s'est passé avec les commandes

```
gitk            # Visualiser l'historique graphiquement
```

et

```
git gui blame hello.c     # voir l'historique de chaque
                          # ligne du fichier hello.c
```

On va maintenant mettre ce « commit » à disposition des autres utilisateurs.

5.2 Fusion de révisions (merge)

SEULEMENT *Bob* fait :

```
git push           # Envoyer les commits locaux dans
                   # le dépôt partagé
```

Pour voir où on en est, les deux équipes peuvent lancer la commande :

```
gitk               # afficher l'historique sous forme graphique
```

ou bien

```
git log            # afficher l'historique sous forme textuelle.
```

À PRESENT, *Alice* peut tenter d'envoyer ses modifications :

```
git push
```

On voit apparaître :

```
To https://forge.univ-lyon1.fr/alice/lifprojet.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to
'https://forge.univ-lyon1.fr/alice/lifprojet.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

2. Les plus curieux d'entre vous pourront lire la page `https://ensiwiki.ensimag.fr/index.php?title=Ecrire_de_bons_messages_de_commit_avec_Git` pour quelques conseils sur la manière de rédiger de bons messages de commit.

L'expression « non-fast-forward » (qu'on pourrait traduire par « absence d'avance rapide ») veut dire qu'il y a des modifications dans le dépôt vers laquelle on veut envoyer nos modifications et que nous n'avons pas encore récupérées. Envoyer nos changements maintenant reviendrait à écraser le travail de Bob, ce que Git nous empêche de faire. Il faut donc fusionner les modifications avant de continuer.

L'utilisateur *Alice* fait donc :

```
git pull
```

Après quelques messages sur l'avancement de l'opération, on voit apparaître :

```
Auto-merging sandbox/hello.c
CONFLICT (content): Merge conflict in sandbox/hello.c
Automatic merge failed; fix conflicts and then commit the result.
```

Ce qui vient de se passer est que *Bob* et *Alice* ont fait des modifications au même endroit du même fichier dans les commits qu'ils ont fait chacun de leur côté (en ajoutant leurs noms sur la même ligne), et Git ne sait pas quelle version choisir pendant la fusion : c'est un conflit, et nous allons devoir le résoudre manuellement. Allez voir `hello.c`.

La bonne nouvelle, c'est que les modifications faites par *Alice* et Bob sur des endroits différents du fichier ont été fusionnés. Quand une équipe est bien organisée et évite de modifier les mêmes endroits en même temps, ce cas est le plus courant : les développeurs font les modifications, et le gestionnaire de versions fait les fusions automatiquement.

En haut du fichier, on trouve :

```
<<<<<<< HEAD
/* Auteurs : Alice et ... */
=======
/* Auteurs : ... et Bob */
>>>>>>> 2483c228b1108e74c8ca4f7ca52575902526d42a
```

Les lignes entre `<<<<<<<` et `=======` contiennent la version de votre commit (qui s'appelle HEAD). les lignes entre `=======` et `>>>>>>>` contiennent la version que nous venons de récupérer par « pull » (nous avions dit qu'il était identifié par la chaîne 2483c22, en fait, l'identifiant complet est plus long, nous le voyons ici).

Il faut alors « choisir » dans `hello.c` la version qui convient (ou même la modifier). Ici, on va fusionner à la main (i.e. avec un éditeur de texte) et remplacer l'ensemble par ceci :

```
/* Auteurs : Alice et Bob */
```

On pourrait aussi utiliser des outils dédiés à la résolution des conflits, comme meld, kdiff3 ou le mode approprié de votre éditeur de texte (`M-x smerge-mode` sous Emacs par exemple).

Si *Alice* fait à nouveau

```
git status
```

On voit apparaître :

```
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit(s) each, respectively.

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)

      both modified:      hello.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Si on n'est pas sur de soi après la résolution des conflits, on peut lancer la commande :

```
git diff    # git diff sans argument, alors qu'on avait
            # l'habitude d'appeler 'git diff HEAD'
```

Après un conflit, Git affichera quelque chose comme :

```
diff --cc hello.c
index 5513e89,614e4b9..0000000
--- a/hello.c
+++ b/hello.c
@@@ -1,5 -1,5 +1,5 @@@
  /* Chacun ajoute son nom ici */
- /* Auteurs : Alice et ... */
 -/* Auteurs : ... et Bob */
++/* Auteurs : Alice et Bob */

  #include <stdio.h>
```

(les '+' et les '-' sont répartis sur deux colonnes, ce qui correspond aux changements par rapport aux deux « commits » qu'on est en train de fusionner. Si vous ne comprenez pas ceci, ce n'est pas très grave !)

Après avoir résolu manuellement les conflits à l'intérieur du fichier, on marque ces conflits comme résolus, explicitement, avec `git add` :

```
$ git add hello.c
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit(s) each, respectively.

Changes to be committed:

      modified:   hello.c
```

On note que `hello.c` n'est plus considéré « both modified » (i.e. « contient des conflits non-résolus ») par Git, mais simplement comme « modified ».

Quand il n'y a plus de fichier en conflit, il faut faire un commit (comme « git pull » nous l'avait demandé) :

```
git commit
```

(Dans ce cas, il est conseillé, même pour un débutant, de ne pas utiliser l'option `-a`, mais c'est un détail)

Un éditeur s'ouvre, et propose un message de commit du type « `Merge branch 'master' of ...` », on peut le laisser tel quel, sauver et quitter l'éditeur.

Nb : si il n'y avait pas eu de conflit, ce qui est le cas le plus courant, « git pull » aurait fait tout cela : télécharger le nouveau commit, faire la fusion automatique, et créer si besoin un nouveau commit correspondant à la fusion.

On peut maintenant regarder plus en détails ce qu'il s'est passé :

```
gitk
```

Pour *Alice*, on voit apparaître les deux « commit » fait par *Bob* et *Alice* en parallèle, puis le « merge commit » que nous venons de créer avec « git pull ». Pour *Bob*, rien n'a changé.

La fusion étant faite, *Alice* peut mettre à disposition son travail (le premier commit, manuel, et le commit de fusion) avec :

```
git push
```

et *Bob* peut récupérer le tout avec :

```
git pull
```

(cette fois-ci, aucun conflit, tout se passe très rapidement et en une commande)

Les deux utilisateurs peuvent comparer ce qu'ils ont avec :

```
gitk
```

ils ont complètement synchronisé leur répertoires. On peut également faire :

```
git pull
git push
```

Mais ces commandes se contenterons de répondre `Already up-to-date.` et `Everything up-to-date`.

5.3 Ajout de fichiers

À present, *Alice* crée un nouveau fichier, `toto.c`, avec un contenu quelconque.
Alice fait

```
git status
```

On voit apparaître :

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      toto.c
nothing added to commit but untracked files present (use "git add" to track)
```

Notre fichier `toto.c` est considéré comme « Untracked » (non suivi par Git). Si on n'ajoute pas `toto.c` au dépôt, il sera ignoré par Git : on n'enregistrera pas son historique, il ne sera pas envoyé aux coéquipiers. Si on veut que `toto.c` soit ajouté au dépôt, il faut l'enregistrer (`git commit` ne suffit pas) : `git add toto.c`
Alice fait à present :

```
git status
```

On voit apparaître :

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file:   toto.c
```

Alice fait à présent ((`-m` permet de donner directement le message de log sur la ligne de commande) :

```
git commit -m "ajout de toto.c"
```

On voit apparaître :

```
[master b1d56e6] Ajout de toto.c
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 toto.c
```

`toto.c` a été enregistré dans le dépôt. On peut publier ce changement :

```
git push
```

Bob fait à présent :

```
git pull
```

Après quelques messages informatifs, on voit apparaître :

```
Fast forward
 toto.c |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 toto.c
```

Le fichier `toto.c` est maintenant présent chez *Bob*.

5.4 Fichiers ignorés par Git

Bob crée à présent un nouveau fichier `temp-file.txt`, puis fait :

```
git status
```

On voit maintenant apparaître :

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      temp-file.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Si *Bob* souhaite que le fichier `temp-file.txt` ne soit pas enregistré dans le dépôt (soit « ignoré » par Git), il doit placer son nom dans un fichier `.gitignore` dans le répertoire contenant `temp-file.txt`. Concretement, *Bob* tappe la commande

```
emacs .gitignore
```

et ajoute une ligne

```
temp-file.txt
```

puis sauve et quitte.

Dans le répertoire `sandbox/` qui vous est fourni, il existe déjà un fichier `.gitignore` qui peut vous servir de base pour vos projets.

Si *Bob* souhaite créer un nouveau `.gitignore` (par exemple, à la racine du projet pour que les règles s'appliquent sur tout le projet), pour que tous les utilisateurs du dépôt bénéficient du même fichier `.gitignore`, *Bob* fait :

```
git add .gitignore
```

Bob fait a nouveau

```
git status
```

On voit apparaître :

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file:   .gitignore
```

Quelques remarques :

- Le fichier `temp-file.txt` n'apparaît plus. C'était le but de la manoeuvre. Une bonne pratique est de faire en sorte que « git status » ne montre jamais de « Untracked files » : soit un fichier doit être ajouté dans le dépôt, soit il doit être explicitement ignoré. Cela évite d'oublier de faire un « git add ».
- En général, on met dans les `.gitignore` les fichiers générés (*.o, fichiers exécutables, ...), ce qui est en partie fait pour vous dans le `.gitignore` du répertoire `sandbox/` (qu'il faudra adapter pour faire le `.gitignore` de votre projet). Les « wildcards » usuels (`*.o`, `*.ad?`, ...) sont acceptés pour ignorer plusieurs fichiers.
- Le fichier `.gitignore` vient d'être ajouté (ou bien il est modifié si il était déjà présent). Il faut à nouveau faire un commit et un push pour que cette modification soit disponible pour tout le monde.

6 Pour conclure...

Bien sûr, Git est bien plus que ce que nous venons de voir, et nous encourageons les plus curieux à se plonger dans le manuel utilisateur et les pages de man de Git pour en apprendre plus. Au niveau débutant, voici ce qu'on peut retenir :

Les commandes

git commit -a enregistre l'état courant du répertoire de travail,

git push publie les commits,

git pull récupère les commits publiés,

git add, git rm et git mv permettent de dire à Git quels fichiers il doit surveiller ("traquer" ou "versionner" dans le jargon),

git status, git diff HEAD pour voir où on en est.

Conseils pratiques

— Ne *jamais* s'échanger des fichiers sans passer par Git (email, scp, clé USB), sauf si vous savez *vraiment* ce que vous faites.
— Toujours utiliser `git commit` avec l'option `-a`.
— Faire un `git push` après chaque `git commit -a`, sauf si on veut garder ses modifications privées. Il peut être nécessaire de faire un `git pull` avant un `git push` si des nouvelles révisions sont disponibles dans le dépôt partagé.
— Faire des `git pull` régulièrement pour rester synchronisés avec vos collègues. Il faut faire un `git commit -a` avant de pouvoir faire un `git pull` (ce qui permet de ne pas mélanger modifications manuelles et fusions automatiques).
— Ne faites jamais un « `git add` » sur un fichier binaire généré : si vous les faites, attendez-vous à des conflits à chaque modification des sources ! Git est fait pour gérer des fichiers sources, pas des binaires.

Quand vous ne serez plus débutants [3], vous verrez que la vie n'est pas si simple, et que la puissance de Git vient de `git commit` sans `-a`, des `git commit` sans `git push`, et de toutes les commandes qui ne sont pas documentées ici (`rebase`, `bisect`, `stash` par exemple) ... mais chaque chose en son temps !

Quand rien ne va plus ...

En cas de problème avec l'utilisation de Git
— Consulter la page `http://ensiwiki.ensimag.fr/index.php/FAQ_Git` sur EnsiWiki. cette page a été écrite pour le projet GL, mais la plupart des explications s'appliquent directement pour vous,
— Demander de l'aide aux enseignants,
— Demander de l'aide sur la mailing-list de Git,
— En cas de problème non-résolu et bloquant, poser la question par email à un enseignant (par exemple votre enseignant référent de projet, ou bien Matthieu Moy).

Dans tous les cas, lire la documentation est également une bonne idée : `http://git-scm.com/documentation` ! Par exemple, le livre numérique de Scott Chacon « Pro Git », simple d'accès et traduit en français : `http://git-scm.com/book/fr/v2`

3. cf. par exemple `http://ensiwiki.ensimag.fr/index.php/Maintenir_un_historique_propre_avec_Git`