

TP: Développement collaboratif décentralisé avec GIT

Sommaire

[Installation de git](#)

[Installation outils graphiques](#)

[Customisation environnement](#)

[Ajouter .bashrc](#)

[Ajouter .gitconfig](#)

[Ajout .gitignore](#)

[Rappels des fondamentaux](#)

[1 Publication de revisions](#)

[2 Branches de developpement](#)

[3 Synchronisation de plusieurs repositories](#)

[4 Modifications publiées, modifications non publiées](#)

Note: Pour convertir ce fichier en pdf : <http://dillinger.io>, Top menu > Utilities > Export as PDF File

Installation de git

```
$ sudo apt-get install git
```

Installation outils graphiques

```
$ sudo apt-get install gitk
$ sudo apt-get install git-gui
$ sudo apt-get install gitg
$ sudo apt-get install meld
```

Customisation environnement

```
$ cd ~
$ git clone https://github.com/magicmonty/bash-git-prompt.git .bash-git-prompt
```

- (master↑3|+1): on branch master, ahead of remote by 3 commits, 1 file changed but not staged
- (status|●2): on branch status, 2 files staged
- (master|+7...): on branch master, 7 files changed, some files untracked
- (master|✖2+3): on branch master, 2 conflicts, 3 files changed
- (master|■2): on branch master, 2 stash entries
- (experimental↓2↑3|✓): on branch experimental; your branch has diverged by 3 commits, remote by 2 commits; the repository is otherwise clean
- (:70c2952|✓): not on any branch; parent commit has hash 70c2952; the repository is otherwise clean

Prompt Structure

By default, the general appearance of the prompt is::

```
(<branch> <branch tracking>|<local status>)
```

The symbols are as follows:

- Local Status Symbols
 - ✓: repository clean
 - ●n: there are n staged files
 - ✖n: there are n unmerged files
 - +n: there are n changed but *unstaged* files
 - ...n: there are n untracked files
 - ■n: there are n stash entries
- Branch Tracking Symbols
 - ↑n: ahead of remote by n commits
 - ↓n: behind remote by n commits
 - ↓m↑n: branches diverged, other by m commits, yours by n commits
- Branch Symbol:

When the branch name starts with a colon ``:`, it means it's actually a hash, not a branch (although it should be pretty clear, unless you name your branches like

Ajouter .bashrc

```
GIT_PROMPT_ONLY_IN_REPO=1
if [ -f ~/.bash-git-prompt/gitprompt.sh ]; then
    . ~/.bash-git-prompt/gitprompt.sh
fi
```

Ajouter .gitconfig

Ajouter dans ~/.gitconfig

```
[user]
  name = Premon Nom
  email = monadresse@mail.fr

[alias]
  co = checkout
  ci = commit
  st = status
  br = branch
  hist = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
  type = cat-file -t
  dump = cat-file -p
```

Ajout .gitignore

Pour chaque copie de travail ajouter dans .gitignore

```
[.]*.s[a-w][a-z]
[.]_s[a-w][a-z]
*.un~
Session.vim
.netrwhist
*~
```

Rappels des fondamentaux

Git est un logiciel de contrôle de versions décentralisé. Contrairement à Subversion ou CVS, qui se base sur un unique dépôt avec lequel se synchronisent une ou plusieurs copies de travail (les copies de travail ne peuvent se synchroniser entre elles : elles doivent passer par le dépôt), chaque copie de travail d'un projet versionné avec Git joue aussi le rôle de dépôt, et il est possible de synchroniser entre elles n'importe quelles copies de travail. De plus, Git permet d'utiliser une ou plusieurs branches de développement et de fusionner entre elles ces branches de développement.

1 Publication de revisions

Nous allons tout d'abord nous intéresser à l'aspect gestionnaire de version de Git: comment enregistrer l'historique des modifications apportées à un projet. Pour initialiser un repository, il convient d'invoquer la commande

```
$ git init monrepo
```

Cette commande initialise un dépôt Git dans le repertoire monrepo (qui est crée si celui-ci n'existe pas). Ce repertoire contient alors à la fois une version de travail (dans monrepo) et un repository Git (dans monrepo/.git). Bien que Git ait été conçu pour gérer du code source, nous allons nous en servir dans ce TP pour gérer des fichiers textes simples, pour nous concentrer sur le fonctionnement de Git plutôt que sur du code.

Question 1.1.

Initialiser un dépôt Git, et créez le fichier burger.txt à la racine du repo qui contient la liste des ingrédients d'un burger (un ingrédient par ligne).

```
steak
salade
tomate
cornichon
fromage
```

Git a plusieurs interfaces utilisateur, la plus complète étant l'interface en ligne de commande (CLI), mais nous allons aussi utiliser Gitg ou Gitk qui sont des interfaces graphiques à Git.

Depuis le repertoire de votre dépôt, lancez Gitg (ou lancez Gitg puis ouvrez votre dépôt).

Gitg a deux onglets History et Commit. Dans l'onglet Commit, on remarque 4 cadres :

- Unstaged qui contient la liste des modifications qui ont été apportées dans le dépôt et qui n'ont pas été sélectionnées pour être committées.
- Staged qui contient la liste des modifications qui ont été apportées et qui ont été sélectionnées pour être committées.
- Changes qui affiche une modification
- Commit message qui contient le message du commit courant

Question 1.2.

Sélectionnez votre fichier burger.txt comme modification à être committée; editez un message de commit, puis commitez. Retournez dans l'onglet History pour observer votre commit.

Question 1.3.

Rajoutez un ingrédient dans burger.txt, puis créez quelques autres sandwich.txt et commitez toutes ces modifications. Regardez l'onglet History, votre deuxième commit doit apparaître.

Question 1.4.

Creez un nouveau sandwich, et modifiez un sandwich existant. Nous allons commiter ces changements avec l'interface en ligne de commande. Commençons par taper

```
$ git commit
```

Que se passe-t-il ? Lisez le paragraphe DESCRIPTION de la page de manuel git-commit(1) (ce que l'on peut faire soit en invoquant man git-commit, soit git help commit). Commitez ces changements. Observer l'arbre de commit dans l'onglet History de Gitg (Attention, il faudra sans doute rafraîchir avec Ctrl+R).

2 Branches de developpement

La partie 1 présentait l'utilisation simple de Git pour créer un historique des modifications. Nous allons maintenant nous concentrer sur la notion de branche. Lors du développement d'un projet, il peut arriver que l'on veuille introduire une nouvelle fonctionnalité dans le projet, sans "casser" le projet. Nous voudrions donc pouvoir basculer instantanément de la version stable du projet à sa version "en développement". C'est ce que nous permettent de faire les branches.

Question 2.1.

Créez une nouvelle branche intitulée "developpement" dans votre repository. Avec Gitg, dans l'onglet History, sélectionnez le dernier commit (attention l'étiquette master représente la branche master), clic-droit puis Create New Branch.

Nous avons donc créé une nouvelle branche, qui est pour l'instant la même que la branche principale. Il est possible de basculer d'une branche à l'autre en cliquant droit sur la branche et sélectionnant Checkout working copy.

Question 2.2.

Dans la branche developpement effectuez quelques modifications (modifications dans les fichiers ou ajout/suppression de fichier), puis commitez-les. Observez ce qu'il se passe dans l'onglet History. (Sélectionnez l'affichage de toutes les branches). En ligne de commande, pour afficher la branche courante, il suffit d'invoquer git branch. Une branche est créée à partir d'une autre au moyen de

```
$ git branch MANOUELLEBRANCHE
```

, et le passage d'une branche à l'autre se fait au moyen de

```
$ git checkout MABRANCHE
```

Question 2.3.

Constatez les différences dans chacune des deux branches. Jusqu'à présent nous avons suivi le scénario simple où il y a une branche de développement, et une branche stable. Supposons que les modifications dans la branche de développement soient finies et que nous voulions nous lancer dans de nouvelles modifications, il convient de synchroniser les deux branches. Cela s'appelle une fusion de branches (merge en anglais).

Question 2.4.

Essayez sous Gitg, de fusionner la branche master avec la branche développement. Est-ce le résultat attendu ? Ce scénario simple ne s'applique pas toujours : pendant que des modifications sont effectuées dans la branche de développement, il peut falloir aussi effectuer des correctifs mineurs dans la branche stable.

Question 2.5.

Effectuez (et commitez) des modifications dans la branche développement et d'autres modifications dans la branche master (attention à ce que ces modifications ne soient pas conflictuelles). Qu'est-ce qui est affiché dans l'onglet History de Gitg ? Il est bien entendu possible de fusionner deux branches avec l'interface en ligne de commande. Lisez le paragraphe DESCRIPTION dans la page de manuel de git-merge(1)

Question 2.6.

Avec l'interface en ligne de commande, fusionnez les branches master et developpement.

Jusqu'à présent, nous n'avons envisagé que des scénarios dans lesquels la fusion des branches est simple, mais il peut y arriver qu'il y ait des conflits, par exemple un même bogue corrigé de manière sensiblement différente dans deux branches différentes.

Question 2.7.

Que se passe-t-il dans ce cas-là ? Essayez d'implémenter ce scénario. Comment Git vous permet-il de résoudre les conflits ? Ecrase-t-il unilatéralement les modifications effectuées dans une branche ?

3 Synchronisation de plusieurs repositories

Jusqu'à présent, nous avons vu quelques fonctionnalités de Git sans nous intéresser son aspect collaboratif. Git permet un travail collaboratif sur un dépôt. C'est-à-dire qu'il est possible de synchroniser entre elles des branches de deux dépôts différents.

Question 3.1.

Creez un nouveau dépôt (avec git init --bare)).

```
$ mkdir $PATH_TO_REPO2
$ git init --bare
```

Ceci initialise un dépôt Git sans copie de travail. Il y a deux façons de synchroniser entre eux deux dépôts :

- soit en récupérant les commits du dépôt distant (pull)
- soit en envoyant des commits vers le dépôt distant (push). Dans le deuxième cas, il faut que le dépôt distant soit un dépôt bare, c'est à dire sans copie de travail.

Question 3.2.

Envoyez les commits de votre premier dépôt vers le second avec les commandes (exécutées depuis votre premier dépôt *monrepo*) :

```
$ git push file:///PATH_TO_REPO2 master:master
```

Cette commande va envoyer la branche master du premier dépôt dans une branche appelée master dans le second dépôt. La seconde va effectuer la même chose avec la branche développement.

```
$ git push file:///PATH_TO_REPO2 developement:developement
```

Observez le résultat en lançant Gitg depuis le second dépôt.

Pour rendre la synchronisation plus intéressante, nous allons utiliser une deuxième copie de travail.

Question 3.3.

Créez une nouvelle copie de travail à partir du dépôt bare:

```
$ git clone file:///PATH_TO_REPO2 copietravail
```

qui crée une copie de travail du second dépôt dans le répertoire *copietravail*.

Question 3.4.

Effectuez quelques modifications dans votre première copie de travail. Propagez ces modifications dans votre troisième dépôt : *copietravail*.

- Envoyez ces modifications dans le second dépôt (avec `git push file:///PATH_TO_REPO2 BRANCHE:BRANCHE`)
- Puis depuis le troisième dépôt, récupérez avec `git pull`, ces modifications depuis le second dépôt. (Comme le troisième dépôt a été créé à partir du second au moyen de `git clone`, il n'est pas nécessaire de préciser ici où Git doit chercher les commits)

Nous avons mis en place avec le second et troisième dépôt le schéma de collaboration avec Git le plus courant : il y a un dépôt qui fait office de dépôt maître, et le troisième dépôt qui peut récupérer et envoyer des commits sur le dépôt maître. Nous allons maintenant nous intéresser à l'accès concurrent à ce dépôt maître.

Question 3.5.

Effectuez des modifications dans le premier dépôt, et envoyez ces modifications dans le second dépôt. Sans synchroniser le troisième dépôt avec le second, effectuez (et committez) des modifications dans le troisième dépôt. Que se passe-t-il maintenant lorsqu'on fait `git pull` dans le troisième dépôt ?

Question 3.6.

`git pull` peut être décomposé en `git fetch` suivi de `git merge`. Réitérez le scénario de la Question précédente mais en faisant `git fetch` au lieu de `git pull`, observez "toutes les branches" du dépôt 3 dans Gitg. Un des aspects fondamentaux de Git est qu'il est décentralisé. Nous avons ici donné un rôle spécial de dépôt central au dépôt 2, mais s'il venait à disparaître, il serait toujours possible de synchroniser entre eux les dépôts 1 et 3.

Question 3.7.

Dans le dépôt 1, nous allons déclarer l'adresse du dépôt 3, nous allons créer pour cela une remote appelée *repo3*.

```
$ git remote add repo3 file:///PATH_TO_REPO3
```

De même, dans le dépôt 3, nous allons créer une remote appelée *repo1* qui pointe vers le premier dépôt.

Question 3.8.

Effectuez (et committez) des modifications dans le dépôt 3 et récupérez-les dans le dépôt 1 au moyen de :

```
$ git fetch repo3
$ git checkout master
$ git merge remotes/repo3/master
```

Nous pouvons simplifier cette démarche en déclarant que la branche master du dépôt 1 "suit" la branche master du dépôt 3 (depuis le dépôt 1), à l'aide de

```
$ git branch master --set-upstream repo3/master
```

Question 3.9.

Effectuez des modifications dans le dépôt 1 puis récupérez-les dans le dépôt 3 au moyen de

```
git pull repo1 master
```

Question 3.10.

Synchronisez entre eux les trois dépôts.

4 Modifications publiees, modifications non publiees

Nous avons vu que les objets que s'échangent les dépôt gits sont des commits. Afin de maintenir une intégrité des arbres de commit, Git utilise des primitives cryptographiques. Chaque commit est en fait signé, en fonction du patch qu'il représente, du nom d'auteur, de la date de création, et aussi de la signature du commit parent (ou des deux parents, dans le cas d'un commit de fusion). Cette signature est un hachage SHA1 de toutes ces informations, et il est possible de se référer à un commit uniquement par cette signature (de la forme f4ccba7ba89d4f6f8f0853056d47912c640a19c1) ou par un préfixe non ambigu de celle-ci (f4ccba7b). Ainsi Git n'appliquera pas un commit ailleurs que sur son père. L'utilisateur de Git pourra vouloir appliquer un commit ailleurs dans l'arbre de commit (par exemple sur une autre branche), il va pour cela devoir créer un nouveau commit (c'est à dire avec un SHA1 différent) mais qui contient les mêmes modifications.

Supposons que nous clonions un dépôt (qu'on appellera repo et nommons commitA le dernier commit sur ce dépôt), et que nous effectuions un commit dans notre copie de travail (commitB dont le père est commitA).

Parallèlement, un autre développeur effectue un commit commitC au dessus du commitA et envoie ce commit dans le dépôt repo.

Il n'est plus possible d'envoyer notre commit commitB, car le dernier commit du depot repo n'est pas commitA, mais commitC.

La stratégie consiste alors à fusionner notre branche locale avec la branche distante, ce qui va créer un commit de fusion, fils de commitB et commitC et d'envoyer ce commit et le commit commitB vers repo.

Nous allons voir qu'une autre stratégie est possible, de demander (avec la commande rebase) à Git de "modifier" le commitB pour que son père soit commitC.

Question 4.1.

Implementez ce scénario, constater que git push renvoie une erreur, puis au lieu d'invoquer git merge, invoquez

```
$ git rebase origin/master
```

(origin/master étant le nom de la branche distante avec laquelle on voudrait normalement effectuer un merge). Git va recréer les commits de votre branche master qui ne sont pas dans la branche master du dépôt repo et va les placer au dessus du dernier commit de la branche master de repo). **Attention, il est très fortement déconseillé de rebase des commits qui ont déjà été publiés**, c'est à dire présent sur un autre dépôt. La Question suivante va donc vous montrer ce qu'il faut éviter de faire.

Question 4.2.

Synchronisez vos 3 dépôts. Dans le dépôt 3, effectuez un commit. Publiez-le dans le depot 2. Dans le dépôt 1 effectuez d'autres commits, récupérez ce commit dans le dépôt 3 à l'aide de

```
$ git fetch repo1
```

(car nous avons déclaré une remote appelée repo1 dans le dépôt 3 qui pointe vers le premier dépôt), puis (dans le dépôt 3), rebasez votre branche master au dessus de la branche repo1/master avec

```
$ git rebase repo1/master
```

Que se passe-t-il si on essaie de merger cette branche avec la branche présente dans le dépôt 2 ?

Un autre cas de modification de commit est avec la sous-commande amend de Git. amend permet d'éditer, de modifier le contenu d'un commit. Supposons qu'on vienne de commiter un commit intitulé orthographe et qu'il corrige des fautes d'orthographes. Supposons qu'une faute ne soit pas corrigée par ce commit, et qu'on ne veuille pas créer un autre commit par dessus, il est possible de modifier le dernier commit avec

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

Question 4.3.

Creez un commit, puis apportez d'autres modifications, et editez le précédent commit au lieu d'en créer un nouveau. Là encore, il est impératif de ne pas éditer un commit qui a déjà été publié, **c'est ce que la Question suivante vous demande de faire et qu'il faut éviter de faire.**

Question 4.4.

Creez un commit, envoyez-le vers un dépôt distant, puis amendez votre commit, synchronisez votre dépôt avec le dépôt distant.