GIT Tutorial

Author : Matthieu FOURNET (remove « YOUR PANTS » to mail me ☺)
URL : http://doc.callmematthi.eu/TOC.html#git
Date : 2019/01/09
This work is licensed under a Creative Commons Attribution – NonCommercial – ShareAlike 4.0 International License.

Contents

1. What is Git made for ?

Git is made to handle text that changes over time : **programs** are text, **scripts** are text, **documentation** and **configuration files** are text, too. Even « **pictures** » can be text (SVG graphics, diagrams internally handeld as XML, ...)

Git is there to help you track :

- **what** has changed
- **when** it was changed
- **who** changed it

and most important :

- **WHY** it changed
- and **what changed at the same time**

This is why Git is not reserved to developers : it can be useful to everybody working with text, changing / sharing and relying on text.

Here are some common use cases you probably have already met :

a. « cpold »

If you have already done (or regularly do) one of these :

```
•  cp someFile someFile.old
•  cp someFile someFile_BACKUP1
•  cp -p someFile someFile.$(date <date options>)
```

⇒ you're doing manually something Git can do (better !) for you.

b. History as a giant header comment

If your scripts start with a giant comment like :

```
# 2012-03-29 bob    initial version
# 2012-04-17 bob    made change X and Y
# 2012-04-28 stuart fix bug Z
# 2013-02-06 kevin  add functionality XY
…
# 2018-11-16 …      ...
```

⇒ you're doing manually something Git can do (better !) for you.

c. Working with others

The « cpold » method above *may* work (albeit unefficient) for a single person. Doing the same with several people modifying files is, at best, extremely complex, adds delays and manual operations, and increases the risk of losing one's work by overwriting files.

⇒ you can NOT collaborate using prehistoric methods. Git has been designed to address this.

2. Git's « Building blocks »

Git works with 3 main blocks :

a. the « working directory »

In this directory are the files you're working on : your program, configuration files, possibly some notes. This is a « normal » directory where you do your work (add / remove / change things). You'll have to explicitely ask Git to track some files before the magic begins ;-)

What's important about this directory is that, since you're tracking the versions of files over time, it can display ANY version of your work : the latest version or any previous version. This is what allows to have a look at previous changes / edit / undo / fix things.

NB : Git won't show you a previous version of a file unless you explicitely ask it to do so. So, 99.9 % of the time, this « working directory » will show the lastest version of your work.

b. the « index » (or « staging area »)

You record changes in Git with a « commit » operation. This is a simple and quick action, but its preparation can be more complex.

Committing is like shipping a parcel : the package you're about to ship is sealed, you just have to write the recipient 's address and leave it at the post office. Before sealing it, you'll have to gather items in a box : you can add one, remove one, replace one, and do anything you like before the package is sealed.

It's the same thing in the « staging area » : this area is an open box where you put some stuff to ship. In the context of « version control », the items to ship are changes made to text files : lines added / removed / changed / moved / …

c. the « .git » directory

A « Git repository » is a regular directory (your « working area ») where Git has initialized some files for its own usage (list of tracked files, history, …). A repository continues « down » into sub-directories, but not up to the parent directories.

Git saves its metadata into a « **.git** » (hidden) directory inside the directory you've chosen as your repository. What's inside this **.git** directory is exclusively Git's internal plumbing, and you mustn't mess with it.

Some googling may suggest to alter files within « .git ». DON'T DO THAT : there are safe ways to do things without risking to corrupt metadata and losing history.

If you remove this **.git** directory (i.e. delete all history), your Git repository instantly transforms into a « normal » directory, in its **current** status… (you shouldn't do that either).

3. Prerequisites

This tutorial contains step-by-step instructions to Git basics. Even though you're not expected to be completely familiar with Git concepts, you should have a look to some documentation first :

- https://ensiwiki.ensimag.fr/images/3/34/Git-slides.pdf
- http://ndpsoftware.com/git-cheatsheet.html
- http://git-scm.com/book/en/v2/Getting-Started-Git-Basics

4. My first Git repository

Let's create a working directory :

```
mkdir -p ~/myFirstGitRepository
cd ~/myFirstGitRepository
```

Let's create a new Git repository there :

```
git init
```

This outputs :

```
Initialized empty Git repository in ~/myFirstGitRepository/.git/
```

```
ls –al
```

outputs :

```
drwxr-x---  3 thomas git  4096 Dec  8 10:38 .
drwxrwxrwt. 9 root   root 4096 Dec  8 10:38 ..
drwxr-x---  7 thomas git  4096 Dec  8 10:38 .git ← this is for metadata
```

5. My first commit

Now let's write a program :

```
echo -e "line1\nline2\nline3" > myProgram
```

What does Git know about it ?

```
git s ← « git s » is an alias for « git status » (see the ~/.gitconfig chapter)
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       myProgram
nothing added to commit but untracked files present (use "git add" to track)
```

Git has detected our new file, but it is not managed by Git so far : « *untracked* ».

Let's ask Git to track it :

```
git a myProgram ← « git a » is an alias for « git add »
```

a. NB : about « tracked » files and « *git add* » :

« Tracking a file » means Git is aware of it. It detects and reports changes made to this file, and you are able to commit such changes.
You don't have to track all files of your working area (for instance, if you track a source code file and a Makefile, tracking the compiled binary would add redundant information to the repository).
The « ***git add*** »command is used to declare a new file to « track », but there's something subtle with this command : « ***git add myFile*** »

- does not politely asks Git to « magically » track **myFile**
- instead, it detects the differences between the current lines of **myFile** and the previous known lines of **myFile**. In that case, ALL the lines of **myFile** are new lines, and Git adds them to the staging area. They are ready to be shipped with the next commit command.

« ***git add myFile*** » is not only for the first time : it is for anytime there are changes, including the first time, where everything is new and appears as a change. The next time you'll change anything to **myFile**, you'll have to run « ***git add myFile*** » again to add the changes to the staging area.

Then :

```
git s
```

```
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   myProgram
```

It's time to commit !!!

```
git co -m "This is my first commit" ← Choosing the right commit message is an ART !
```

```
[master (root-commit) 487f1da] This is my first commit
 1 files changed, 3 insertions(+), 0 deletions(-)
```

```
 create mode 100644 myProgram
```

Let's have a look at the « history » (i.e. : *logs*) :

```
git log
```

```
commit 487f1da203c469a0ed47f02a6c579b0d56a74577
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 13:28:38 2014 +0100

    This is my first commit
```

487f1da203c469a0ed47f02a6c579b0d56a74577 is the « *commit ID* ». It is computed from commit data + header information hashed with SHA1. It can be considered as a unique ID.

6. A less basic commit

Let's improve our program with a new file for a function :

```
echo "function1" > myFunction
```

Then we'll edit **myProgram** to « include » our function file :

```
include myFunction
line1
line2
line3
```

What's up, Git ?

```
git s
```

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   myProgram
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       myFunction
no changes added to commit (use "git add" and/or "git commit -a")
```

- **myProgram** has changed since the latest commit
- **myFunction** is *untracked* (i.e. we've not yet asked Git to *track* it)

What are the differences in **myProgram** ?

```
git d myProgram ← « git d » is an alias for « git diff »
```

```
diff --git a/myProgram b/myProgram
index 83db48f..c9d212e 100644
--- a/myProgram
+++ b/myProgram

@@ -1,3 +1,4 @@
+include myFunction ← this line was added : notice the leading '+'

 line1
 line2
 line3
```

Let's « *stage* » the differences :

```
git a myProgram
```

```
(returns nothing)
```

```
git a myFunction
```

```
(returns nothing)
```

```
git s
```

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

Changes are « *staged* » (i.e. they are stored in the « *staging area* » a.k.a the « *index* »). Let's register them into our repository with a *commit* :

```
git commit -m "New function file 'included' by myProgram"
```

```
[master bff43d6] New function file 'included' by myProgram
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 myFunction
```

NB : the first line fields are :

- `master` : the current branch (more about branches later ;-)
- `bff43d6` : leading characters of the commit ID (see below)
- the commit message.

History of my commits?

```
git log
```

```
commit bff43d60116098a599d10443b9cfc6fcfe0b577e
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 15:05:38 2014 +0100

    New function file 'included' by myProgram

commit 487f1da203c469a0ed47f02a6c579b0d56a74577
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 13:28:38 2014 +0100

    This is my first commit
```

7. Here comes the magic !

My files are now handled by Git, which means I can recover any previous version of every single file.

a. Discard all changes

Let's imagine we've made some changes into **myProgram** :

```
echo "this is bad code I will regret later" > myProgram
```

Now, **myProgram** is broken and won't work anywore ☹ since it looks like this :

```
this is bad code I will regret later
```

But I can recover it :

```
git checkout myProgram
```

```
(returns nothing)
```

myProgram now looks like :

```
include myFunction
line1
line2
line3
```

b. Undelete files

Deleting a file is also considered as a change by Git, and this can be un-done. Let's try this :

```
rm myProgram
ls
```

```
myFunction
```

Recover :

```
git checkout myProgram
```

myProgram is back :

```
include myFunction
line1
line2
line3
```

\o/

8. Git must be aware of every change !

Our little program has improved, and **myFunction** now contains several functions :

```
function1
function2
function3
```

Exercise : edit **myFunction** to add functions as shown above, then commit your changes.

a. Renaming a file

Everything is fine, except that the file name « **myFunction** » doesn't look appropriate anymore. But if we rename the file without telling Git :

```
mv myFunction myFunctions
git s
```

```
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    myFunction
#
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       myFunctions
no changes added to commit (use "git add" and/or "git commit -a")
```

Git considers we have :

- Deleted **myFunction**
- Created **myFunctions**

This means the whole history of modifications made to **myFunction** will stop here, and a new blank history starts for **myFunctions**. This is **not** what we want to do.
So let's rename back our file and see the right method to proceed :

```
mv myFunctions myFunction
git mv myFunction myFunctions ← « git mv » tells git we're renaming files
ls -1
```

```
myFunctions ← the file has been renamed \o/
myProgram
```

```
git s
```

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

Then we can commit :

```
git co myFunction myFunctions -m "Rename my functions file"
```

```
[master bf2d1a8] Rename my functions file
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename myFunction => myFunctions (100%)
```

NB : In this special case, we have to explicitely specify the names of the objects to commit to `git commit` :

```
git commit <object(s) to commit> –m "commit message"
```

As well as other changes, renaming a file is monitored by Git and appears in the history :

```
git log
```

```
commit f6b40a605dada3f1aa84a8a5b324ffaf35f72f5e
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 17:32:10 2014 +0100

    Rename my functions file

commit 7709e0d6590a4a7ff8812416267a570f5a04e006
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 16:28:32 2014 +0100

    Add function2 and function3

...
```

b. Deleting a file

The same method applies when deleting files tracked by Git : don't delete anything without telling Git :

Do not :
```
rm <fileToBeDeleted>
```

But do instead :
```
git rm <fileToBeDeleted>
```

Then *stage* this and *commit* it ;-)

9. Commit error

Ooops : I forgot to update the « *include* » statement in **myProgram** : so far, I « *include myFunction* » instead of « *include myFunction**s*** ». 2 solutions :

- Update **myProgram** and make an « *ooopscommit* » (works but ugly !)
- Update **myProgram** and *amend* the previous *commit*. We'll do it this way !

Let's start by fixing our code so that **myProgram** looks like :
```
include myFunctions
line1
line2
line3
```

Differences ?
```
git d myProgram
```

```
diff --git a/myProgram b/myProgram
index c9d212e..05ede74 100644
--- a/myProgram
+++ b/myProgram
@@ -1,4 +1,4 @@

-include myFunction

+include myFunctions

 line1
 line2
 line3
```

Let's add these differences to the index :
```
git a myProgram
```

Then, let's amend our previous commit :
```
git co --amend
```

This will open the previous commit message in the default text editor (mostly Vi or Nano). We can then edit this message so that it gives a good description of the changes made by both commits. Save and exit shows :
```
[master 69ce360] Rename my functions file
 2 files changed, 1 insertions(+), 1 deletions(-)
```

```
 rename myFunction => myFunctions (100%)
```

Let's check :

```
git log
```

```
commit 69ce3607a6d2a2dd8b3746d6d0212462e1bb3d3a
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 17:32:10 2014 +0100

    Rename my functions file

commit 7709e0d6590a4a7ff8812416267a570f5a04e006
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 16:28:32 2014 +0100

    Add function2 and function3
```

NB : there's only one commit about renaming files.

To have a detailled view of what changes were made by this commit), run :

```
git show <commitIdBefore>..<commitId>
```

In our example :

```
git show
7709e0d6590a4a7ff8812416267a570f5a04e006..69ce3607a6d2a2dd8b3746d6d0212462e1bb3d3a
```

```
commit 69ce3607a6d2a2dd8b3746d6d0212462e1bb3d3a
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Mon Dec 8 17:32:10 2014 +0100

    Rename my functions file

diff --git a/myFunction b/myFunction
deleted file mode 100644
index feb2fb2..0000000
--- a/myFunction
+++ /dev/null
@@ -1,4 +0,0 @@

-function1
-function2
-function3


-
diff --git a/myFunctions b/myFunctions
new file mode 100644
index 0000000..feb2fb2
--- /dev/null
+++ b/myFunctions
@@ -0,0 +1,4 @@

+function1
+function2
+function3


+
diff --git a/myProgram b/myProgram
index c9d212e..05ede74 100644
--- a/myProgram
+++ b/myProgram
@@ -1,4 +1,4 @@

-include myFunction
```

```
+include myFunctions

 line1
 line2
 line3
```

NB : When dealing with commit IDs, Git accepts shortened versions of a commit ID as long as it refers to a single commit from the current repository. The command line above is equivalent to :

```
git show 7709..69ce
```

(Of course, you will have different commit IDs when trying this, but you get the idea ;-)
Maybe you have already noticed the short commit ID displayed while completing a commit :

```
[master 69ce360] Rename my functions file
```

Means :

```
[<branchName> <shortCommitId>] <commitMessage>
```

10.Branches

a. Creating a new branch

Let's imagine our program as evolved so that it looks like :

- **myFunctions** :

```
function1
function2
function3
...
functionN
```

- **myProgram** :

```
include myFunctions
line1 using function1
line2 using function2
a very interesting line of code
# don't change the line below
doNotTouchThisIsMagic
line3
call functionN
```

Exercise : edit both files and commit changes (in a single commit !)

The program has been released to end-users and is now in its « production » step. This phase involves some bug fixes, as well as minor changes. In the meantime, developers are considering rewriting most of their functions for performance / code cleaning / new 3rd-party API / any reason. This means they must maintain both versions and be able to switch from one to each other quickly, at any time, if possible.

Git *branches* are made for this. You may have already noticed :

```
git s
```

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

By default, we're already using a branch called « *master* ». We've developed our program on this branch, made commits, renamed files, … Let's decide the *master* branch will be dedicated to the « production » version of the program, and let's create a new branch for the refactoring. We'll call it *refactoring* :

```
git branch refactoring
```

```
(returns nothing)
```

```
git s
```

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

NB : We've not yet « jumped » to the new branch.

```
git checkout refactoring
```

```
Switched to branch 'refactoring'
```

Done : we're on our new *refactoring* branch.

To check which branch you're on :

```
git branch
```

```
  master
* refactoring
```

The * highlights the current branch.

Now, the code refactoring takes place. Here's the listing of **myFunctions** :

```
function1
clean_function2
clean_function3
new_function4
...
functionN
```

Then :

```
git a myFunctions
git co -m "Refactor of my functions"
git log
```

```
commit e6ec617cc0c1d6c9fc77d904a44b3a3a3398226e
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 12:21:58 2014 +0100

    Refactor of my functions

commit 2ff1f594be6afc4babc859a8a6f69fd8339d745b
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 11:52:24 2014 +0100

    Little Program becomes bigger.

(previous commits not shown here ;-)
```

b. Jumping from branch to branch

ALERT : a bug was found on the production version of the program. We must stop working right now on the refactoring and start fixing this bug.
To do so, let's jump back to our *master* branch :

```
git checkout master
```

```
Switched to branch 'master'
```

Exercise : can you remember another way to get the current branch name ?

Let's have a look at **myFunctions** . It should look like :

```
function1
function2
function3
...
functionN
```

It's the version we've released to production ☺.

Let's fix the bug now. **myProgram** looks like :

```
include myFunctions
line1 using function1
line2 using function2
a very interesting line of code
# don't change the line below
doNotTouchThisIsMagic
myVariable = 42 ← the best bugfix ever !
line3
call functionN
```

Exercise : Edit myProgram and commit changes so that `git log` returns :

```
commit 7c61d22f2ca5f905044024a9ab9eacf846961226
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 12:34:38 2014 +0100

    Bug fix

commit 2ff1f594be6afc4babc859a8a6f69fd8339d745b
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 11:52:24 2014 +0100

    Little Program becomes bigger.

(previous commits not shown here ;-)
```

NB :

- The commit IDs will be different ;-)
- The commit right before the bug fix is the « *Little Program becomes bigger.* » commit. The « *Refactor of my functions* » commit we just made is invisible because it lies on another branch.

c. Branches, what then ?

This situation can not last forever :

- If the refactoring fails, we may decide to delete the « *refactoring* » branch, improve our TODOs/DON'Ts development guidelines and move on from this new experience.
- If the refactoring works, we have to « mix » the changes made on the « *refactoring* » branch into the « *master* » branch (having our « bug fix » changes). This way, we'll produce a new improved production version.

We'll continue with the « refactoring works » hypothesis. But for the sake of learning, here is what we'd have to do in the other situation : deleting the « *refactoring* » branch :

1. Like in the real life, you should not try to delete the branch you're sitting on ! So move to another branch first (any branch will fit) :
```
git checkout master
```

2. Delete the branch :
```
git branch -d refactoring
```

Ok, time has come to *mix* both branches. What we want to do now is to « *take* » all the changes we made on the files of the « *refactoring* » branch and « *add* » them to the files of the « *master* » branch. This operation is called a « *git merge* ».

NB : be careful when merging : there is a « *source* » and a « *destination* » branch. Switching them may lead to unexpected results ;-)

To merge « *refactoring* » (source) into « *master* » (destination) :

Jump to the destination branch :
```
git checkout master
```
```
Switched to branch 'master'
```

Or (if you were already there) :
```
Already on 'master'
```

Merge the source branch (*refactoring*) on the current branch (*master*) :
```
git merge refactoring
```
```
Merge made by recursive.
 myFunctions |    5 +++--
 1 files changed, 3 insertions(+), 2 deletions(-)
```

Then delete the merged branch :
```
git branch -d refactoring
```
```
Deleted branch refactoring (was e6ec617).
```

NB : deleting the merged branch is not mandatory. It is possible to continue developing on a branch such as « *refactoring* », and merging it several times. It all depends on the context of each project.

d. Branches : the end

Let's have a look at our source code :

myFunctions :

```
function1
clean_function2
clean_function3
new_function4
...
functionN
```

This is our refactoring.

myProgram :

```
include myFunctions
line1 using function1
line2 using function2
a very interesting line of code
# don't change the line below
doNotTouchThisIsMagic
myVariable = 42
line3
call functionN
```

And we even have our bugfix !

Let’s have a look at our history :

```
git log
```

```
commit de17d94e8cf9fdd87d1b9085ed09c2cf7580522a
Merge: 7c61d22 e6ec617
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 16:12:10 2014 +0100

    Merge branch 'refactoring'

commit 7c61d22f2ca5f905044024a9ab9eacf846961226
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 12:34:38 2014 +0100

    Bug fix

commit e6ec617cc0c1d6c9fc77d904a44b3a3a3398226e
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 12:21:58 2014 +0100

    Refactor of my functions

commit 2ff1f594be6afc4babc859a8a6f69fd8339d745b
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 11:52:24 2014 +0100

    Little Program becomes bigger.

(previous commits not shown here ;-)
```

The log shows the « *bug fix* » and « *refactor* » commits, and another one for the *merge* itself.

11.Remotes

So far, we've created a local repository and played with it. Fine, but Git is essentially about collaborating with others and sharing code with them.

Let's imagine a new developer (called *Bob)* joins the team. He will have his own local repository, and both repositories (Bob's and mine) will exchange data about the program we're developing. From my point of view, Bob's local repository will be a *remote* repository. So is mine for him.

a. Let's create a remote repository

Let's create Bob's repository. It can actually be anywhere on the file system, or even on a different machine (and generally in **/home/bob/development/**). Anything Git does locally in this example can be done over http / SSH / …, but this is beyond the scope of this tutorial.

```
mkdir -p ~/bob/repository
cd ~/bob/repository
```

Since Bob is joining the development team, he needs to get the full repository :

- the source code (a.k.a. the *working copy*, found in **~/myFirstGitRepository**)
- the history (i.e. the *metadata*, found in **~/myFirstGitRepository/.git**)

We're going to provide him with this data thanks to `git clone` :

```
git clone ~/myFirstGitRepository/ .  ← don't forget the trailing "."
```

```
Initialized empty Git repository in ~/bob/repository/.git/
```

NB : git clone expects 2 parameters : a source repository and a destination directory. If you omit the destination, it will create, in the current directory, a new directory named after the repository name. Running :

```
git clone ~/myFirstGitRepository/  ← no trailing "." this time
```

Will create **~/bob/repository/myFirstGitRepository/** .

Why Git states it « *initialized an empty repository* » when cloning is still mysterious because this new repository is not empty at all. Let's make sure our files are there :

```
ls -l
```

```
total 8
-rw-r----- 1 thomas git  70 Dec  9 17:37 myFunctions
-rw-r----- 1 thomas git 184 Dec  9 17:37 myProgram
```

As well as the history :

```
git log
```

```
commit de17d94e8cf9fdd87d1b9085ed09c2cf7580522a
Merge: 7c61d22 e6ec617
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 16:12:10 2014 +0100

    Merge branch 'refactoring'

(previous commits not shown here ;-)
```

b. Having a look at this « remote » thing

`git clone` is very convenient, because as soon as we clone a repository, this repository (the source repository) is configured as a *remote* of the newly created repository (destination repository).

Let's check :

```
git remote -v
```

```
origin  /home/thomas/myFirstGitRepository (fetch)
origin  /home/thomas/myFirstGitRepository (push)
```

This indicates that the directory we created at the very beginning of this tutorial (**/home/thomas/myFirstGitRepository**) is a *remote* repository that can be used to *fetch* and to *push* data. It also states that « *origin* » is an alias for this *remote* repository (this is a default setting, as well as « *master* » for the default branch).

c. Let developers share code

Let's see how the code is shared between developpers. Back to the 1st repository (**/home/thomas/myFirstGitRepository**), we're going to improve our program.

Exercise : update the source code so that :

```
git d
```

Returns :

```
diff --git a/myProgram b/myProgram
index 02dcb3b..a3586c5 100644
--- a/myProgram
+++ b/myProgram
@@ -6,4 +6,6 @@ a very interesting line of code
 doNotTouchThisIsMagic
 myVariable = 42
 line3
+
+some more very interesting code
```

Commit so that the log says :

```
commit 2c6424f4f7da32fa9b18dfa3ed6663dd35739150
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 18:12:57 2014 +0100

    To infinity and beyond
```

Let's now move to Bob's repository and check the log :

```
cd ../bob/repository/
git log
```

```
commit de17d94e8cf9fdd87d1b9085ed09c2cf7580522a
Merge: 7c61d22 e6ec617
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 16:12:10 2014 +0100

    Merge branch 'refactoring'
```

The latest commit made by Thomas is not there ☹ (there's some magic within Git, but don't expect too much ;-)

What if we asked politely ?

```
git pull -f origin master
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/thomas/myFirstGitRepository
 * branch            master     -> FETCH_HEAD
First, rewinding head to replay your work on top of it...
Fast-forwarded master to 2c6424f4f7da32fa9b18dfa3ed6663dd35739150.
```

Let's check the log :

```
git log
```

```
commit 2c6424f4f7da32fa9b18dfa3ed6663dd35739150
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 18:12:57 2014 +0100

    To infinity and beyond

commit de17d94e8cf9fdd87d1b9085ed09c2cf7580522a
Merge: 7c61d22 e6ec617
Author: Thomas ANDERSON <thomas.anderson@metacortex.com>
Date:   Tue Dec 9 16:12:10 2014 +0100

    Merge branch 'refactoring'

(previous commits not shown here ;-)
```

\o/

Let's have a look at **myProgram** :

```
include myFunctions
line1 using function1
line2 using function2
a very interesting line of code
# don't change the line below
doNotTouchThisIsMagic
myVariable = 42
line3
call functionN

some more very interesting code ← this is the change Thomas just made
```

12. Final advice

a. Help on git commands :

There is a *man* page for every git command. It is available with :

```
git <command> --help
```

b. Architecture, branches, remotes and naming

The way we organize our Git repositories (local, remotes, branches, aliases, ...), and the names given to them is completely up to the developers team. Git itself has very few restrictions on the workflow organization, even though good practices exist.

13.Sample ~/.gitconfig :

Here is my ~/.gitconfig

Feel free to adapt it to your needs !

```
[alias]
	a	= add
	ap	= add --patch
	b	= branch
	co	= commit
	d	= diff
	ds	= diff --staged
	l	= ls-files
	s	= status
#	s	= status --untracked-files=no

[branch]
	autosetuprebase = always  # turn "git pull" into "git pull --rebase"
automatically

[color]
	diff   = auto
	status = auto
	branch = auto

[user]
	email  = thomas.anderson@metacortex.com
	name   = Thomas ANDERSON
```