Version control with Git

June 4, 2015

Part 1: Local version control (only one Git repository)

- The init command
- The add, commit and log commands
- How it works: working copy, staging area (index) and database.
- The diff and status commands
- The log command
- The checkout command
- More on commits
- The SHA-1
- Git branches
- The merge command
- What Git branches are

Part 2: Centralized (à la cvs/svn) version control

- The –bare option of the init command
- The remote command
- The push command
- The fetch command
- Remote branches
- Pushing a (feature) branch
- Tracking branch
- Visualizing branches

Part 3: Distributed workflow

1 Git, a distributed Version Control System

Version control means keeping track of code evolution by recording the code's state after each meaningful change. This is useful:

- To cancel a non-working modification
- To perform regression tests / continuous integration

The database where the different states of the code are recorded can be:

- Local: the project has only one developer
- Centralised: it is available on a server but developers do not receive it when they get the project's sources (cvs, svn).
- Distributed: every developer that has the sources also has the full code history (git, mercurial, darcs).

Version Control Manager:

- Source code is frequently **committed** into Git database, and each **commit** can be retrieved, shared with team.
- Keep, search your code history.
- Develop software in team efficiently.

Distributed:

- Unlike **SVN** which is **centralized**, **Git** is **distributed**. It means that Git does not require to use one central repository, but multiple ones may be used.
- When one downloads source code from a Git repository, it creates a new Git repository, with the full database. There is no conceptual difference between the two repositories.
- Offline work possibility
- Multiple possible workflows to collaborate with other developers.

Some terminology:

- Repository: the vesion-controlled project and optionally the database containing the project's history.
- Commit: one record of the code's state in the project's history.
- Branch: maintain several versions of the project in parallel

Git makes it easy to work with **branches**.

- Branches are easy to create, merge and destroy.
- Creating temporary branches to develop a feature is encouraged.

Git has a few core concepts that must be understood.

- Without knowing these core concepts, using Git is frustrating and painful.
- Knowing them, using Git is powerful and easy.

Thanks to Git's simplicity for creating new repositories and managing branches, a workflow adapted to your team may be chosen. For example:

- working with a central repository and contributing into branches (small private teams);
- working with forks and contributing with pull requests (large teams with external contributors).

This presentation deals with the core concepts of Git, so as to make its adoption easier.

2 Local version control (only one Git repository)

We start working on a single repository.

In this section, we will learn the core concepts of Git:

- commits,
- staging area (index),
- branches.

Start by creating a new empty directory to experiment with Git:

```
In [1]: from os import makedirs, chdir, curdir, walk, sep
        from os.path import expanduser, isdir, abspath, join, relpath, basename
        from shutil import rmtree

        workdir = expanduser('~/git-training')

        # Remove possible existing working, and starts with a fresh one.
        if isdir(workdir):
            rmtree(workdir)
        makedirs(workdir)

        def changedir(directory):
            """Set up a directory relative to workdir"""
            directory = abspath(join(workdir,directory))
            if not isdir(directory):
                makedirs(directory)
            chdir(directory)
            print("We are in directory " + directory)

        # The directory that will contain the Git repository
        changedir('repo')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/repo
```

2.1 The init command

A Git **repository** is created in the current directory using the **init** command.

This will create a `.git` hidden directory, where Git stores its database.

```
In [2]: %%bash
        git init
        ls -la
```

```
Initialized empty Git repository in /nas/home3/f/froger/git-training/repo/.git/
total 12
drwxr-xr-x 3 froger sed 4096 Jun  4 13:18 .
drwxr-xr-x 3 froger sed 4096 Jun  4 13:18 ..
drwxr-xr-x 7 froger sed 4096 Jun  4 13:18 .git
```

You can now start developing. For example, we may write “First Line” in a file called `foo.txt`. But in practice, we would probably want to write some real source code.

```
In [3]: %%bash
        echo "First line" > foo.txt
```

2.2 The add, commit and log commands

To record (commit) the previous changes to Git's database, do as follows:

```
In [4]: %%bash
        git add foo.txt
        git commit -m "Initial commit."
```

```
[master (root-commit) cb1a8e7] Initial commit.
 1 file changed, 1 insertion(+)
 create mode 100644 foo.txt
```

The **add** command tells Git to track changes in the file `foo.txt` and says that the current content of this file should be commited by the next **commit** command.

The **commit** command itself then records (commits) all the added changes to git's database and associates a few metadata to this set of changes.

The argument of -m is a commit message, that is, a description of the commited change. The commit message is mandatory. If -m is not given, the **commit** command will open an editor where the commit message should be typed.

To make sure things have been properly commited, one may use the **log** command:

```
In [5]: %%bash
        git log
```

```
commit cb1a8e7e1afde0d21aa75e2188c0fdff1042fb2f
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:31 2015 +0200

    Initial commit.
```

As can be seen, the metadata associated with the commit includes its author. However, the name or e-mail address printed at the moment may not look so pretty. Here is how to improve this for future commits:

```
In [6]: %%bash
        # git config --global user.name "Prénom Nom"
        # git config --global user.email prenom.nom@inria.fr
```

It is even possible to fix the authorship of our previous commit:

```
In [7]: %%bash
        #git commit --amend --reset-author
```

And let's make sure this actually worked:

```
In [8]: %%bash
        git log
```

```
commit cb1a8e7e1afde0d21aa75e2188c0fdff1042fb2f
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:31 2015 +0200

    Initial commit.
```

Note: we used the –**local** flag here so that the configuration applies only to the current repository, but one can also use the –**global** flag to make the coniguration the default for all repositories, given that it can then be overloaded in each repository.

2.3 How it works: working copy, staging area (index) and database.

Woking copy and database are conceps which are well known in other revision control systems. To these, Git adds a third zone, the staging area. It is an intermediary place where changes go before being commited to Git's database.

Having this third area may seem odd at first, but it turns out to be useful e.g. to seperate commits, as we will see.

Suppose we add two lines to foo.txt:

```
In [9]: %%bash
        echo "Second line" >> foo.txt
        echo "Third line" >> foo.txt
```

But then we realise that these two lines really represent two distinct changes and should thus be commited separately.

Git makes it possible to achieve this thanks to its index or staging area, like this:

```
In [10]: %%bash
         #git add -p foo.txt
```

And choose 'e' to edit the hunk, then remove the line
+Third line
so that the only line starting wih a + symbol "Second line".

To make sure only the second line has been added to the index and will thus be commited, use the **diff** command as follows:

```
In [11]: %%bash
         git diff --cached
```

Before continuing, notice how we now have three different versions of foo.txt:

- One in the working copy (3 lines)
- One in the index (2 lines)
- One in the database (1 line)

Let's commit what has been staged:

```
In [12]: %%bash
         git commit -m "Second commit"
```

```
# 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:   foo.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Let's make sure the commit has worked:

```
In [13]: %%bash
         git log
```

```
commit cb1a8e7e1afde0d21aa75e2188c0fdff1042fb2f
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:31 2015 +0200

    Initial commit.
```

And note that the staging area is now empty:

```
In [14]: %%bash
         git diff --cached
```

We can now commit our second change to foo.txt:

```
In [15]: %%bash
         git add foo.txt
         git commit -m "Third commit"
```

```
[master d3d41b9] Third commit
 1 file changed, 2 insertions(+)
```

Two remarks are due here:

1. The example we have jsut seen to understand why the staging area is useful is quite artificial. It is however rather imortant, because the situation it describes can happen quite a lot in practice. For instance, suppose that while adding a feature to a program one discovers typos in the existing code. The new feature and the typo fixups could for sure be commited together, but doing two distinct commits is considered better practice because it gives a cleaner history (In particular, should the feature be removed later, that could be achieved without loosing the typo fixups.)

In such a situation, the -p flag to the add command turns out to be especially useful. Moreover, since the changes happen most of the time in different hunks (regions), it will be easier to use the interactive **add** in such situations than in the one above, since it will not require any manual hunk edition as before.

2. The three areas that have just been introduced (working copy, staging area and commit database) are of crucial importance. Indeed, almost all git commands either manipulate one of these areas or transfer content between two of them and understanding Git in terms of how the commands work on areas turns out to be especially helpful (if not fundamental) in practice.

Moreover, for one specific command, its arguments may change the areas it affects. As an example, git commit transfers content from the staging area to the database, but with the -a argument, the same command will transfer all the uncommited (and unstaged) changes directly from the working copy to the database and leave the staging area unmodified.

Exercise: can you explain what **log** and **add** do in terms of the three areas?

2.4 The diff and status commands

Modify the `foo.txt` file, and observe the outputs of the **diff** and **status** commands

```
In [16]: %%bash
         echo 'Fourth line' >> foo.txt
```

```
In [17]: %%bash
         git diff
```

```
diff --git a/foo.txt b/foo.txt
index 6da4d3e..5028ae5 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,3 +1,4 @@
 First line
 Second line
 Third line
+Fourth line
```

```
In [18]: %%bash
         git status
```

```
# 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:   foo.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Stage the file, and observe the new outputs of the **diff** and **status** commands

```
In [19]: %%bash
         git add foo.txt
```

```
In [20]: %%bash
         git diff
```

```
In [21]: %%bash
         git diff --cached
```

```
diff --git a/foo.txt b/foo.txt
index 6da4d3e..5028ae5 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,3 +1,4 @@
 First line
 Second line
 Third line
+Fourth line
```

```
In [22]: %%bash
         git status
```

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

Commit the file.

```
In [23]: %%bash
         git commit -m 'Add fourth line to foo.txt'
```

```
[master 0b6dacd] Add fourth line to foo.txt
 1 file changed, 1 insertion(+)
```

2.5 The log command

The **log** command prints an history of all the commits.

```
In [24]: %%bash
         git log
```

```
commit 0b6dacd6d0792b096a1d075614f100bb5d7afcd2
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:32 2015 +0200

    Add fourth line to foo.txt

commit d3d41b9db307ba8d5698d23c4eefa5376b6dac06
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:32 2015 +0200

    Third commit

commit cb1a8e7e1afde0d21aa75e2188c0fdff1042fb2f
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:31 2015 +0200

    Initial commit.
```

```
In [25]: %%bash
         git log -p
```

```
commit 0b6dacd6d0792b096a1d075614f100bb5d7afcd2
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:32 2015 +0200

    Add fourth line to foo.txt

diff --git a/foo.txt b/foo.txt
index 6da4d3e..5028ae5 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,3 +1,4 @@
 First line
 Second line
 Third line
+Fourth line

commit d3d41b9db307ba8d5698d23c4eefa5376b6dac06
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:32 2015 +0200

    Third commit

diff --git a/foo.txt b/foo.txt
index 9649cde..6da4d3e 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1 +1,3 @@
 First line
```

```
+Second line
+Third line

commit cb1a8e7e1afde0d21aa75e2188c0fdff1042fb2f
Author: David Froger <david.froger@inria.fr>
Date:   Thu Jun 4 13:18:31 2015 +0200

    Initial commit.

diff --git a/foo.txt b/foo.txt
new file mode 100644
index 0000000..9649cde
--- /dev/null
+++ b/foo.txt
@@ -0,0 +1 @@
+First line
```

A common practice when writing commit messages is to start with a one-line description of the commit, optionally followed by a longer description which may be split into several paragraphs.

Another thing one may do when writing commit messages is to explain more why the change is done than the change itself, since the change can be figured out by studying the patch itself.

2.6 The checkout command

The **checkout** command updates files in the working tree to match the version in the index or the specified tree. For example:

```
In [26]: %%bash
         git checkout master^

Note: checking out 'master^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at d3d41b9... Third commit
```

Will ask Git to set-up the working copy according to the content of Git's database at commit master^, namely one commit before master as indicated by the ^ postfix operator.

Let's verify:

```
In [27]: %%bash
         cat foo.txt

First line
Second line
Third line
```

And now let's restore the working copy as it was before this checkout:

```
In [28]: %%bash
         git checkout master

Previous HEAD position was d3d41b9... Third commit
Switched to branch 'master'
```

And let's verify that this worked, too:

```
In [29]: %%bash
         cat foo.txt

First line
Second line
Third line
Fourth line
```

2.7 More on commits

To take advantage of all the powerful features of Git, it is important to understand what a **commit** actually is.

The first thing to know is that Git has stored in its database 1 commit object,for each commit, each commit object containing a complete version of the `foo.txt` file.

- For the first commit, Git has stored in its database a commit object containing `First line`.
- For the second commit, Git has stored in its database a commit object containing not the difference between the two versions, but the whole file: `First line` (hence duplicated in Git's database), and `Second line`.
- After the second commit, `First line` **is** duplicated in the 2 different commit objects in Git's database.

Note that for performance, Git has the ability to efficiently compress its database and handle differences only (especially during network transfer), but the model is to store the whole content of files for each commit, as opposed to some other revision control systems which only store differences.

This yields a very simple model. A commit contains the directories and files we have commited (called `tree` and `blob` in Git), plus some metadata.

In Git, a commit object contains:

- At least one parent commit (except for the initial commit)
- The (root) tree (which itself contains trees and blobs).
- The commit message.
- The author.
- The commit date.

2.8 The SHA-1

With Git it is not possible to assign integer numbers to commits in a sequential way as is done in svn for instance, because Git's distributed nature and branches make the very notion of linear sequence vanish.

Git uses the **SHA-1** cryptographic hash function to identify each object (**commit**, **tree**, **blob**) with a hash value. Such a hash value may look like the following one: `0e1e060688a560015614cf7ec4b77d8a0df07c2f`.

The hash value is computed from the object's content. It is very unlikely that two diferent commit objects have the same **SHA-1** hash value. The likelihood of such collisions is so low that it is generally considered

to be 0, meaning that in practice it is considered that having same SHA-1 hash and being the same commit are equivalent propositions.

Each hash value identifies only one commit. It also identifies all the directories and files that belong to the commit. Note that parent commits are also part of the commit: two commits sharing the same files and directories, but with different commit parents, will have different hash values.

Note:

- if two developpers create exactly the same commit on two different computers, the hash value will be the same,
- we know that two commits are different by only comparing their hash values,
- hash value are very fast to compute: if a whole tree in a commit has not changed, Git does not have to recompute its hash again.

2.9 Git branches

Suppose we now want to try developing a new feature in our code, while continuing our previous work on `foo.txt`.

Git encourages creating a branch for this.

A branch is created with the **branch** command, followed by a branch **name**:

```
In [30]: %%bash
         git branch bar
```

Without any argument, the **branch** command lists all the branches and marks the current one with an asterisk (git status also displays the current branch).

```
In [31]: %%bash
         git branch
```

```
bar
* master
```

The **checkout** command allows you to switch to another branch:

```
In [32]: %%bash
         git checkout bar
         git branch
```

```
* bar
  master

Switched to branch 'bar'
```

It is very important to remember that git branch b creates branch b but does not change the current branch. This is similar to Unix's mkdir command which creates a new directory without changing the current directory to the one it just created. To continue the analogy with Unix commands, git checkout b changes the current branch in the same way cd changes the current directory.

It is however common when creating a branch that the intention is to switch to that branch right after it has been created and this is what the git checkout -b command does. In other words, what had been achieved in two steps before (namely git branch bar and git checkout bar) can be achieved with just the following command: git checkout -b bar.

Now, let us develop different things in the two branches:

```
In [33]: %%bash
         echo 'First line' > bar.txt
         git add bar.txt
         git commit -m 'First line of bar.txt'

         echo 'Second line' >> bar.txt
         git add bar.txt
         git commit -m 'Second line of bar.txt'

         git checkout master
         echo "Fifth line" >> foo.txt
         git add foo.txt
         git commit -m  'Fifth line of foo.txt'

         echo "Sixth line" >> foo.txt
         git add foo.txt
         git commit -m 'Sixth line of foo.txt'

         git checkout bar
         echo 'Third line' >> bar.txt
         git add bar.txt
         git commit -m 'Third line of bar.txt'
```

```
[bar b166c88] First line of bar.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bar.txt
[bar e88fd6a] Second line of bar.txt
 1 file changed, 1 insertion(+)
[master 5791763] Fifth line of foo.txt
 1 file changed, 1 insertion(+)
[master 5f34e3a] Sixth line of foo.txt
 1 file changed, 1 insertion(+)
[bar 0880e14] Third line of bar.txt
 1 file changed, 1 insertion(+)

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

2.10 The merge command

We merge the work of the two branches. More specifically, we merge the **bar** branch into the **master** branch

```
In [34]: %%bash
         git checkout master
         git merge bar
```

```
Merge made by the 'recursive' strategy.
 bar.txt |    3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 bar.txt

Switched to branch 'master'
```

Because there is no conflict, the merge is performed automatically. In case of confict (same lines of a file modified in both branches):

- the merge operation stops,
- the developper edits the conflicting files to solve the conflict,
- the developper commits the merged files.

2.11 What Git branches are

Edit the file ~ and add the content:

```
[alias]
  gr = log --graph --full-history --all --color --pretty=tformat:"%x1b[31m%h%x09%x1b[32m%d%x1b[0m%x20%s
```

This adds a useful **gr** command to Git, that displays a colored graph of the branches.

```
In [35]: %%bash
         git gr
```

```
*   e8971a6         (HEAD, master) Merge branch 'bar' (David Froger)
|\
| * 0880e14         (bar) Third line of bar.txt (David Froger)
| * e88fd6a         Second line of bar.txt (David Froger)
| * b166c88         First line of bar.txt (David Froger)
* | 5f34e3a         Sixth line of foo.txt (David Froger)
* | 5791763         Fifth line of foo.txt (David Froger)
|/
* 0b6dacd         Add fourth line to foo.txt (David Froger)
* d3d41b9         Third commit (David Froger)
* cb1a8e7         Initial commit. (David Froger)
```

All the commits form a chain, in which each commit is linked to its parent(s).

Creating a branch means having two commits with the same parent, while merging means creating a commit with two parents.

We can now give a simple definition of a branch: a symbolic name that points to a commit with no children.

Two special branches are:

- **master**, the original branch when a repository is created. That's a branch like the others.
- **HEAD**, the current branch, which is updated after each commit (like $PWD in Unix shells).

When a commit is checked out that is not the end of a branch (has children), it is said that the repository is in "detached head" mode. In that state, it is possible to create commits which will be linked to tehe one that has been checked out, but it must be kept in mind that no symbolic name (apart from HEAD) will be associated with the last commit, so when HEAD moves to a different branch the repo wil have a branch with no symbolic name associated to its tip and which may hence be garbage-collected later by Git. It is however possible and easy to associate a symbolic name with a commit at any time with the git branch command.

Note: with this knowledge on commits and branches, some Git features not demonstrated here will be easy to understand: - rebase - fast-forward - tags (symbolic names which don't move, as opposed to branches)

3 Exercice

During the exercice, experiment with `git log`, `git status`, `git gr`, etc.

0- Initialize an empty Git repository.

1- Create a script `main.py` with the following content, and commit it.

```
In [36]: #!/usr/bin/env python

         def greet():
             print("Hello world!")

         greet()
```

```
Hello world!
```

2- Modify the the script, commit it

```
In [37]: #!/usr/bin/env python

         def greet(name):
             print("Hello %s!" % name)

         greet("Alice")
```

```
Hello Alice!
```

3- Modify and commit the script again.

```
In [38]: #!/usr/bin/env python
         import sys

         def greet(name):
             print("Hello %s!" % name)

         if len(sys.argv) > 1:
             greet(sys.argv[1])
         else:
             sys.stderr.write("Usage: %s NAME\n" % sys.argv[0])
             sys.exit(1)
```

```
Hello -f!
```

4- In a branch `format_name`, modify and commit the script:

```
In [39]: #!/usr/bin/env python
         import sys

         def greet(name):
             print("Hello %s!" % name.capitalize())

         if len(sys.argv) > 1:
             greet(sys.argv[1])
         else:
             sys.stderr.write("Usage: %s NAME\n" % sys.argv[0])
             sys.exit(1)
```

```
Hello -f!
```

5- In the `master` branch, modify the script and commit:

```
In [40]: #!/usr/bin/env python
         import sys
```

```
        def greet(name):
            print("Hello %s! How are you?" % name)

        if len(sys.argv) > 1:
            greet(sys.argv[1])
        else:
            sys.stderr.write("Usage: %s NAME\n" % sys.argv[0])
            sys.exit(1)
```

```
Hello -f! How are you?
```

6- Merge the `format_name` branch. You will have to resolve a conflict, and the commit

4 Centralized (à la cvs/svn) version control

Now that we have learned how to work with a single Git repository, we will learn how to send/receive commits between two Git repositories

In this section, we will assume a workflow with two developpers: Alice and Bob. Both of them have their own repository on their computer:

```
- Alice's repository A on her computer,
- Bob's repository B on his computer.
```

and a central repository on a computer which both Alice and Bob can communicate with:

```
- central repository C on a "server".
```

For simplicity, we will demonstrate the commands on the same machine, using Git **file://** protocol. However, Git commands would be **exactly the same**, but using instead the **ssh://** or **https://** protocols.

Note that configuring a "server" machine to host a Git repository and managing user permissions, backup, availability, Web views of the repository, etc. is not easy. Forges like **Inria's GForge**, **GitHub**, **Bitbucket**, **Gitorious** should be preferred.

```
In [41]: changedir('')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training
```

4.1 The –bare option of the init command

We start by creating a central repository.

There is a subtlety. Suppose we create a git repository in $\sim$ and that someone else edits files in this Git repository.

It is possible that someone else sends commits (in Git, this is called **push**) to this repository, which would be stored in $\sim$. Then Git's database and working copy would differ.

To avoid this situtation, Git provides the **–bare** option to **init**. It creates a Git repository, but without a working copy. Nobody can **commit** directly into this repository, but only **push** commits.

Never **push** commits to a Git repository that is not **bare**, to avoid inconsistencies with its working copy.

By convention, **bare** repositories are suffixed with **.git**, even if it is not necessary.

```
In [42]: %%bash
         git init --bare central.git
         ls -l central.git
```

```
Initialized empty Git repository in /nas/home3/f/froger/git-training/central.git/
total 28
drwxr-xr-x 2 froger sed 4096 Jun  4 13:18 branches
-rw-r--r-- 1 froger sed   66 Jun  4 13:18 config
-rw-r--r-- 1 froger sed   73 Jun  4 13:18 description
-rw-r--r-- 1 froger sed   23 Jun  4 13:18 HEAD
drwxr-xr-x 2 froger sed 4096 Jun  4 13:18 hooks
drwxr-xr-x 2 froger sed 4096 Jun  4 13:18 info
drwxr-xr-x 4 froger sed 4096 Jun  4 13:18 objects
drwxr-xr-x 4 froger sed 4096 Jun  4 13:18 refs
```

Alice clones the central repository:

```
In [43]: %%bash
         git clone file://$PWD/central.git alice

Cloning into 'alice'...

warning: You appear to have cloned an empty repository.
```

Alice enters her Git repository and configures her name and email for that repository (we assume here she didn't do it at the global level, to keep all the examples self-contained):

```
In [44]: changedir('alice')

We are in directory /home/ROCQ/sedrocq/froger/git-training/alice

In [45]: %%bash
         git config --local user.name "Alice"
         git config --local user.email alice@inria.fr
```

4.2 The remote command

Without argument, the *remote* command lists the **remote** repositories Git knows about:

```
In [46]: %%bash
         git remote

origin
```

When the central repository has been cloned, Git has given it the name **origin**, by convention.
Later, we will learn how to register additionnal remote repositories.
Alice works and commits into her repository:

```
In [47]: %%bash
         echo 'First line' > foo.txt
         git add foo.txt
         git commit -m 'First line of foo.txt'

         echo 'Second line' >> foo.txt
         git add foo.txt
         git commit -m 'Second line of foo.txt'

[master (root-commit) b4182d9] First line of foo.txt
 1 file changed, 1 insertion(+)
 create mode 100644 foo.txt
[master 6a5b517] Second line of foo.txt
 1 file changed, 1 insertion(+)
```

4.3 The push command

Now, Alice wants to send her commits to the central repository, so that Bob can get them.

To do so, she uses the **push** command, whose arguments are:

```
git push <remote_name> <local_branch>:<remote_branch>
```

The remote_name is **origin**, Alice pushes the **master** branch of her repository to the **master** branch of the central repository:

```
In [48]: %%bash
         git push origin master:master
```

```
To file:///nas/home3/f/froger/git-training/central.git
 * [new branch]      master -> master
```

Bob now clones the central repository too, enters it and configures his name and email:

```
In [49]: changedir('')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training
```

```
In [50]: %%bash
         git clone file://$PWD/central.git bob
```

```
Cloning into 'bob'...
```

```
In [51]: changedir('bob')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/bob
```

```
In [52]: %%bash
         git config --local user.name "Bob"
         git config --local user.email bob@inria.fr
```

Doing so, Bob fetches Alice's work.

```
In [53]: %%bash
         git log
```

```
commit 6a5b51790c06eb9405d665e51814218017e4da73
Author: Alice <alice@inria.fr>
Date:   Thu Jun 4 13:18:34 2015 +0200

    Second line of foo.txt

commit b4182d9b3880504ed53d0f60bbd89af90d01c77a
Author: Alice <alice@inria.fr>
Date:   Thu Jun 4 13:18:34 2015 +0200

    First line of foo.txt
```

Bob makes some changes, commits and pushes:

```
In [54]: %%bash
         echo "Third line" >> foo.txt
         git add foo.txt
         git commit -m "Third line to foo.txt"
         git push origin master:master
```

```
[master 20c9479] Third line to foo.txt
 1 file changed, 1 insertion(+)
```

```
To file:///nas/home3/f/froger/git-training/central.git
   6a5b517..20c9479  master -> master
```

4.4 The fetch command

```
In [55]: changedir('alice')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/alice
```

Alice wants to get Bob's commits. She uses the **fetch** command, which donwloads all the commits of all branches from a remote repository.

```
In [56]: %%bash
         git fetch origin
```

```
From file:///nas/home3/f/froger/git-training/central
   6a5b517..20c9479  master     -> origin/master
```

4.5 Remote branches

The git **branch** command lists all branches of Alice's repository, also known as local branches:

```
In [57]: %%bash
         git branch
```

```
* master
```

But where are the **central** repository branches which have just been fetched?
Adding the **-a** option to the **branch** command reveals them:

```
In [58]: %%bash
         git branch -a
```

```
* master
  remotes/origin/master
```

`remotes/central/master.git` is called a **remote branch**.
A **remote branch** is a read-only branch that reflects the state of a branch of a remote repository. If the branch changes on the remote repository, use **fetch** again to refresh it.
Note: to see only remote branches rather than all branches one can use the -r flag instead of -a:

```
In [59]: %%%bash
         git branch -r
```

```
origin/master
```

To get all commits of `remotes/central/master` **remote branch** into the **master** branch, merge it:

```
In [60]: %%bash
         git merge remotes/origin/master
```

```
Updating 6a5b517..20c9479
Fast-forward
 foo.txt |    1 +
 1 file changed, 1 insertion(+)
```

Note: **fetch** and **merge** operations can be accomplished in one command, **pull**:

```
git pull <remote_name> <remote_branch>:<local_branch>
```

4.6 Pushing a (feature) branch

While Alice and Bob are working on the master branch, Alice wants to develop an experimental feature. She creates a branch for this, and works in it:

```
In [61]: %%bash
         git checkout -b exp

         echo "First line" > bar.txt
         git add bar.txt
         git commit -m 'First line of bar.txt'

         echo "Second line" >> bar.txt
         git add bar.txt
         git commit -m "Second line in bar.txt"
```

```
[exp badb1a5] First line of bar.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bar.txt
[exp 73ba4df] Second line in bar.txt
 1 file changed, 1 insertion(+)

Switched to a new branch 'exp'
```

Alice then pushes her branch to a similarly named branch of the central repository:

```
In [62]: %%bash
         git push origin exp:exp
```

```
To file:///nas/home3/f/froger/git-training/central.git
 * [new branch]      exp -> exp
```

4.7 Tracking branch

At the same time, Bob has worked on the master branch:

```
In [63]: changedir('bob')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/bob
```

```
In [64]: %%bash
         echo "Third line" >> foo.txt
         git add foo.txt
         git commit -m "Third line in foo.txt"

         echo "Fourth line" >> foo.txt
         git add foo.txt
         git commit -m "Fourth line in foo.txt"
```

```
[master bb2e0af] Third line in foo.txt
 1 file changed, 1 insertion(+)
[master 4cd3a70] Fourth line in foo.txt
 1 file changed, 1 insertion(+)
```

Bob wants to see Alice's work on the `exp` branch. He fetches all branches of the central repository:

```
In [65]: %%bash
         git fetch
         git branch -r
```

```
origin/HEAD -> origin/master
  origin/exp
  origin/master

From file:///nas/home3/f/froger/git-training/central
 * [new branch]      exp        -> origin/exp
```

Bob has remote branch `central/exp`, but how to work with it?
Adding the –**track** option to the **checkout** command makes Git create a **tracking branch**:

```
In [66]: %%bash
         git checkout --track origin/exp

Branch exp set up to track remote branch exp from origin.

Switched to a new branch 'exp'
```

A tracking branch is our local copy of a remote branch. Unlike the remote branch, we have write access to it.
Tracking branches can also be used to call the **pull** and **push** commands without arguments.
Note that Alice and Bob can work on the `exp` branch without changing anything to the `master` branch.

- if `exp` was not a good idea, the branch can be dropped,
- if `exp` is a good idea, it may be merged into the `master` branch.

4.8 Visualizing branches

Bob helps Alice to develop the `exp` branch by making a new commit:

```
In [67]: %%bash
         echo "Third line" >> bar.txt
         git add bar.txt
         git commit -m "Third line in bar.txt"

[exp 6e69132] Third line in bar.txt
 1 file changed, 1 insertion(+)
```

At this point, it is instructive to visualize the different branches:

```
In [68]: %%bash
         git gr

* 6e69132         (HEAD, exp) Third line in bar.txt (Bob)
* 73ba4df         (origin/exp) Second line in bar.txt (Alice)
* badb1a5         First line of bar.txt (Alice)
| * 4cd3a70         (master) Fourth line in foo.txt (Bob)
| * bb2e0af         Third line in foo.txt (Bob)
|/
* 20c9479         (origin/master, origin/HEAD) Third line to foo.txt (Bob)
* 6a5b517         Second line of foo.txt (Alice)
* b4182d9         First line of foo.txt (Alice)
```

We note that:

```
- Bob's master branch has 2 more commits than the central one.
- Bob's exp branch has 1 more commit that the central one.
```

The verbose option of **branch** is useful to see tracking branches:

```
In [69]: %%bash
         git branch -avv
```

```
* exp                   6e69132 [origin/exp: ahead 1] Third line in bar.txt
  master                4cd3a70 [origin/master: ahead 2] Fourth line in foo.txt
  remotes/origin/HEAD   -> origin/master
  remotes/origin/exp    73ba4df Second line in bar.txt
  remotes/origin/master 20c9479 Third line to foo.txt
```

Bob pushes the commits of the two branches.

```
In [70]: %%bash
         git push origin exp:exp
         git push origin master:master
```

```
To file:///nas/home3/f/froger/git-training/central.git
   73ba4df..6e69132  exp -> exp
To file:///nas/home3/f/froger/git-training/central.git
   20c9479..4cd3a70  master -> master
```

```
In [71]: %%bash
         git gr
```

```
* 6e69132         (HEAD, origin/exp, exp) Third line in bar.txt (Bob)
* 73ba4df         Second line in bar.txt (Alice)
* badb1a5         First line of bar.txt (Alice)
| * 4cd3a70         (origin/master, origin/HEAD, master) Fourth line in foo.txt (Bob)
| * bb2e0af         Third line in foo.txt (Bob)
|/
* 20c9479         Third line to foo.txt (Bob)
* 6a5b517         Second line of foo.txt (Alice)
* b4182d9         First line of foo.txt (Alice)
```

Finally, the `exp` branch is merged into master. The merge commit is pushed to the central repository, and the exp branch is deleted:

```
In [72]: %%bash
         git checkout master
         git merge exp
         git push origin master:master
         git branch -d exp
         git push origin :exp
         git gr
```

```
Merge made by the 'recursive' strategy.
 bar.txt |    3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 bar.txt
Deleted branch exp (was 6e69132).
*   4a38f5c         (HEAD, origin/master, origin/HEAD, master) Merge branch 'exp' (Bob)
|\
| * 6e69132         Third line in bar.txt (Bob)
| * 73ba4df         Second line in bar.txt (Alice)
| * badb1a5         First line of bar.txt (Alice)
```

```
* | 4cd3a70          Fourth line in foo.txt (Bob)
* | bb2e0af          Third line in foo.txt (Bob)
|/
* 20c9479         Third line to foo.txt (Bob)
* 6a5b517         Second line of foo.txt (Alice)
* b4182d9         First line of foo.txt (Alice)
```

```
Switched to branch 'master'
To file:///nas/home3/f/froger/git-training/central.git
   4cd3a70..4a38f5c  master -> master
To file:///nas/home3/f/froger/git-training/central.git
 - [deleted]         exp
```

5 Distributed workflow

Committing to a central repository is okay for small teams where developpers know and trust each other. For larger projects, though, it is blocking and dangerous to give write access to the project's main repository to an external contributor.

Since Git is a distributed revision control system, clloning a repository means creating a copy of that repository that has exactly as much information. This opens the road to a workflow different rom the previous one, where each developer has a public repository from which everybody can read. When this developer wants to share code, he/she pushes the code to be shared to his/her public repository and informs the maintainers of the project that there is some code available that may beintegrated to the project. The maintainers review the code and, if they find it useful, integrate it to the project's main code base.

This workflow is made possible by the way Git has been designed. The GitHub platform provides two mechanisms (fork and pull request) that help developers adopting this workflow, but the workflow itself relies only on Git features and does not require GitHub to be implemented.

When a developper wants to contribute to a project, he/she forks the original bare repository. It means that the developper gets his own bare repository. In this repository, he develops a feature in a branch. When the work is done, he asks an administrator to pull the feature branch from his repository to the master branch of the project's main repository.

In GitHub terminology, this is called a **pull request**.

Let us see this in practice by adding **Emma** as a developper to our previous example.

```
In [73]: changedir('')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training
```

Emma starts by forking the central repository to her own bare repository.

Note: this is a functionnality provided out of the box by GitHub.

Then Emma clones her bare public repository.

```
In [74]: %%bash
         git clone --bare central.git central-emma-fork.git

         git clone central-emma-fork.git emma
```

```
Cloning into bare repository 'central-emma-fork.git'...
done.
Cloning into 'emma'...
done.
```

```
In [75]: changedir('emma')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/emma
```

```
In [76]: %%bash
         git config --local user.name "Emma"
         git config --local user.email emma@inria.fr
```

Emma creates a topic branch, works on it, and pushes it to her public repository:

```
In [77]: %%bash
         git checkout -b baz

         echo "First line" > baz.txt
         git add baz.txt
         git commit -m 'First line in baz.txt'

         echo "Second line" >> baz.txt
         git add baz.txt
         git commit -m 'Second line in baz.txt'

         git push origin baz:baz
```

```
[baz 1ae59f8] First line in baz.txt
 1 file changed, 1 insertion(+)
 create mode 100644 baz.txt
[baz 569eae6] Second line in baz.txt
 1 file changed, 1 insertion(+)

Switched to a new branch 'baz'
To /nas/home3/f/froger/git-training/central-emma-fork.git
 * [new branch]      baz -> baz
```

```
In [78]: changedir('alice')
```

```
We are in directory /home/ROCQ/sedrocq/froger/git-training/alice
```

The next step for Emma is to ask Alice to get the **baz** branch from **central-emma-fork** pulled into the **master** branch of **central**.

Note: GitHub provides functionnality to request a pull, and review the associated code.

Alice fetches Emma's bare repository. To do this, she first adds Emma's bare repository to the list of her remote repositories.

```
In [79]: %%bash
         git remote add emma file://$HOME/git-training/central-emma-fork.git
         git fetch emma
```

```
From file:///home/ROCQ/sedrocq/froger/git-training/central-emma-fork
 * [new branch]      baz        -> emma/baz
 * [new branch]      master     -> emma/master
```

At this point, Alice and Emmma can interact with each other to add more commits to the `baz` branch. When her work is done, Alice merges it to `master`, and `pushes`.

```
In [80]: %%bash
         git checkout master
         git merge emma/baz
         git push origin master:master
```

```
Updating 20c9479..569eae6
Fast-forward
 bar.txt |    3 +++
 baz.txt |    2 ++
 foo.txt |    2 ++
 3 files changed, 7 insertions(+)
 create mode 100644 bar.txt
 create mode 100644 baz.txt

Switched to branch 'master'
To file:///nas/home3/f/froger/git-training/central.git
   4a38f5c..569eae6  master -> master
```

6 Exercice

This exerice continues on the previous one, and re-use its Git repository. This will be the repository of the first developer.

0- Create a bare repository for the first developer.
1- Push existing commits to this repository.
2- Set up a bare repository and a fork for a second developer.
3- The second developer creates a feature branch, and pushes it to its bare repository.
4- The first developer get the feature branch of the second developer, and pushes it to its bare repository.

7 References

http://www.git-scm.com/docs

http://githowto.com

"Version Control with Git, Powerful tools and techniques for collaborative software development"
By Jon Loeliger, Matthew McCullough

HitHug let's you learn Git through a nice game: https://github.com/gazler/githug

Another game: https://github.com/jlord/git-it