git

November 23, 2014

0.1 Introduction

Git is a **distributed Version Control Manager**.

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 possibiilty - Multiple possible workflows to collaborate with other developers.

Git make it easy to work with **branches**. - Branches are easy to create, merge and destroy. - Create temporary branches to develop a topic is encouraged.

Git has some core concepts that must be understood. - Without knowing these core concept, using Git is frustring and painfull. - Knowing them, using Git is powerfull and easy.

Because of Git easiness to create new repositories and manage 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 withg the core concept of Git, so as to make its adoption easier.

0.2 Local version control (only one Git repository)

We start working on a repository alone.

In this sections, we will learn the core concepts of Git: - staging area, - commits, - branches.

Start by creating a fresh 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 contains the Git repository
changedir('repo')
```

```
We are in directory /home/david/git-training/repo
```

0.2.1 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 /home/david/git-training/repo/.git/
total 12
drwxrwxr-x 3 david david 4096 Nov 23 22:01 .
drwxrwxr-x 3 david david 4096 Nov 23 22:01 ..
drwxrwxr-x 7 david david 4096 Nov 23 22:01 .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
```

0.2.2 The staging area and add commmand

We ask Git to track changes in the file `foo` using **add** command

```
In [4]: %%bash
        git add foo.txt
```

This command does two things: - Because this the first time with use **add** on foo.txt, it tells Git to track changes of this file. - The second thing is more subtle. **add** put the file and its content to what is called the **staging area**, which is in the **.git** directory. Later, we will use **commit** command to commit `foo.txt` into Git database. What Git put into its database is not the content of `foo.txt` file, located in `~/git-training/repo` directory: it puts the content of the file as it is in the **staging_area**, which may differ.

Let us see it in action on our example.

Until now, we have: - written `First line` in `foo.txt`, - run **add** command on `foo.txt`.

So: - `foo.txt` in the working directory contains `First line`. - `foo.txt` in the staging area contains `First line` as well.

Now suppose that we add another line in `foo.txt`.

```
In [5]: %%bash
        echo "Second line" >> foo.txt
```

Now: - `foo.txt` in the working directory contains `First line` and `Second line`. - `foo.txt` in the staging area still contains `First line` only.

If we commit `foo.txt` into the Git database now, using **commit** command, it will commit the file as it is in the staging area.

To update the staging area, we use **add** command again, which copies `myfile.txt` from the working directory into the staging area.

```
In [6]: %%bash
        git add foo.txt
```

Using the staging area, we can choose and prepare exactly what to commit. For example, we may want to commit unrelated changes in our files in two separate commits. The staging area allows us to do it easily.

0.2.3 Commit command

Now, `foo.txt` version of the staging area will be committed into the Git database.
A message is written with the commit, to help remember its meaning.

```
In [7]: %%bash
        git commit -m 'Two lines in the new foo.txt file.'
```

```
[master (root-commit) c2ff06f] Two lines in the new foo.txt file.
 1 file changed, 2 insertions(+)
 create mode 100644 foo.txt
```

0.2.4 The diff and status command

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

```
In [8]: %%bash
        echo 'Third line' >> foo.txt
```

```
In [9]: %%bash
        git diff
```

```
diff --git a/foo.txt b/foo.txt
index 7d91453..6da4d3e 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,2 +1,3 @@
 First line
 Second line
+Third line
```

```
In [10]: %%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 output of **diff** and **log** commands

```
In [11]: %%bash
         git add foo.txt
```

```
In [12]: %%bash
         git diff
```

```
In [13]: %%bash
         git diff --cached
```

```
diff --git a/foo.txt b/foo.txt
index 7d91453..6da4d3e 100644
--- a/foo.txt
```

```
+++ b/foo.txt
@@ -1,2 +1,3 @@
 First line
 Second line
+Third line
```

```
In [14]: %%bash
         git status
```

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

        modified:   foo.txt
```

Commit the file.

```
In [15]: %%bash
         git commit -m 'Add third line to file.txt'
```

```
[master 0aee827] Add third line to file.txt
 1 file changed, 1 insertion(+)
```

0.2.5 The log command

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

```
In [16]: %%bash
         git log
```

```
commit 0aee82772d3700b6b5f0bd186b3cac03a93d0563
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:14 2014 +0100

    Add third line to file.txt

commit c2ff06f748bce77292d29000e1bf359242aa7de4
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:13 2014 +0100

    Two lines in the new foo.txt file.
```

```
In [17]: %%bash
         git log -p
```

```
commit 0aee82772d3700b6b5f0bd186b3cac03a93d0563
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:14 2014 +0100

    Add third line to file.txt

diff --git a/foo.txt b/foo.txt
index 7d91453..6da4d3e 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,2 +1,3 @@
```

```
 First line
 Second line
+Third line

commit c2ff06f748bce77292d29000e1bf359242aa7de4
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:13 2014 +0100

    Two lines in the new foo.txt file.

diff --git a/foo.txt b/foo.txt
new file mode 100644
index 0000000..7d91453
--- /dev/null
+++ b/foo.txt
@@ -0,0 +1,2 @@
+First line
+Second line
```

0.2.6 Checkout command

checkout command restores a previous commit. We are in a `detached HEAD` state, which will be explained later in `branch` section.

```
In [18]: %%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 c2ff06f... Two lines in the new foo.txt file.
```

```
In [19]: %%bash
         cat foo.txt
```

```
First line
Second line
```

```
In [20]: %%bash
         git checkout master
```

```
Previous HEAD position was c2ff06f... Two lines in the new foo.txt file.
Switched to branch 'master'
```

```
In [21]: %%bash
         cat foo.txt
```

```
First line
Second line
Third line
```

0.2.7 Git commits

To take advantages of all the powerfull features of Git, it is important to underdand what actually a **commit** is.

The first thing to know is that Git has stored in its database 3 commits that contains the full 3 versions of the file `foo.txt`. - At the first commit, Git has stored in its database a commit containing `First line`. - At the second commit, Git has stored in its database a commit containing not the difference of the two versions, but the whole file: `First line` (duplicated in Git database) and `Second line`. - After the third commit, `First line` **is** duplicated in the 3 different commits in Git database.

Note that for performance, Git has the ability to efficiently compresses its database and stores differences only (especially during network communication), but the model is to store the whole content of files of each commit.

This yields to 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 contains: - One parent commit (or more). - The (root) tree (which itself contains trees and blobs). - The commit message. - The author. - The commit date.

0.2.8 The SHA-1

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

The hash value is computed from the object content. It is almost impossible that **SHA-1** gives the same hash value for two different contents. The risk of collision is almost zero, and we consider it to be zero.

Each hash value identifies only one commit. It also identifies all the directories and files that makes the commit. Note that parent commits are also part of the commit: two commits sharing 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 is very fast to compute: if a whole tree in a commit has not changed, Git does not have to recompute it.

0.2.9 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 **branch** command, followed by a branch **name**:

```
In [22]: %%bash
         git branch bar
```

Without any argument, **branch** command lists all the branches and marks the current one with an asterix.

```
In [23]: %%bash
         git branch
```

```
bar
* master
```

checkout command allows you to switch to another branch:

```
In [24]: %%bash
         git checkout bar
         git branch
```

```
* bar
  master
```

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

Now, let us develop something in the two branches:

```
In [25]: %%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 "Fourth line" >> foo.txt
         git add foo.txt
         git commit -m  'Fourth line of foo.txt'

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

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

```
[bar f12e434] First line of bar.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bar.txt
[bar 7a4f584] Second line of bar.txt
 1 file changed, 1 insertion(+)
[master e7d4bfb] Fourth line of foo.txt
 1 file changed, 1 insertion(+)
[master 48b94cd] Fifth line of foo.txt
 1 file changed, 1 insertion(+)
[bar 29e75d7] Second line of bar.txt
 1 file changed, 1 insertion(+)

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

0.2.10 The merge command

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

```
In [26]: %%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'
```

Bescause 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 conflicted files, and solves the conflict, - the developper commits the merged files.

0.2.11 What Git branches are

Edit the file `~/.gitconfig` and add the content:

```
[alias]
  graph = log --graph --full-history --all --color --pretty=tformat:"%x1b[31m%h%x09%x1b[32m%d%x1b[0m%x2
```

This will provide Git with a usefull **gr** command, that displays a colored graph of the branches.

```
In [27]: %%bash
         git graph
```

```
*   8c8f8ec         (HEAD, master) Merge branch 'bar' (David Froger)
|\
| * 29e75d7         (bar) Second line of bar.txt (David Froger)
| * 7a4f584         Second line of bar.txt (David Froger)
| * f12e434         First line of bar.txt (David Froger)
* | 48b94cd         Fifth line of foo.txt (David Froger)
* | e7d4bfb         Fourth line of foo.txt (David Froger)
|/
* 0aee827         Add third line to file.txt (David Froger)
* c2ff06f         Two lines in the new foo.txt file. (David Froger)
```

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

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 : their are just a reference/pointer/name for a given commit.

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.

The situation of a **detached HEAD** seen above means restoring an old commit, while leaving HEAD pointing to the commit we where in.

Note: with this knowledge on commits and branches, some Git features not demonstrated here will be easy to understand: - rebase - fast-forward - tag

0.3 Centralized version control

Now 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 consider the workflow of two developpers: - Alice, - Bob,

each one having its own repository on its computer: - Alice repository A on Alice's computer, - Bob repository B on Bob's computer.

and a central repository on a computer that 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 users permissions, backup, availability, Web views of the repository, etc, is not easy. Forges like **GitHub**, **Bitbucket**, **Gitorious** shoudl be preferred.

```
In [28]: changedir(workdir)
```

```
We are in directory /home/david/git-training
```

0.3.1 Init –bare command

We start by creating a central repository.

There is a subtlety. Suppose we create a git repository in `~/git-training/central` and that someone else edits files in this Git repository.

It is possible that someone else sends commits (in Git, this is call **push**) in this repository, which would be stored in `~/git-training/central/.git`. Then Git database and the files in the directory would be inconsistent.

To avoid this situtation, Git provides –**bare** options to **init**. It creates a Git repository, but without working directory. Nobody can **commit** directly into this repository, but only **send** commits via **pushes**.

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

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

```
In [29]: %%bash
         git init --bare central.git
         ls -l central.git
```

```
Initialized empty Git repository in /home/david/git-training/central.git/
total 32
-rw-rw-r-- 1 david david   23 Nov 23 22:01 HEAD
drwxrwxr-x 2 david david 4096 Nov 23 22:01 branches
-rw-rw-r-- 1 david david   66 Nov 23 22:01 config
-rw-rw-r-- 1 david david   73 Nov 23 22:01 description
drwxrwxr-x 2 david david 4096 Nov 23 22:01 hooks
drwxrwxr-x 2 david david 4096 Nov 23 22:01 info
drwxrwxr-x 4 david david 4096 Nov 23 22:01 objects
drwxrwxr-x 4 david david 4096 Nov 23 22:01 refs
```

Alice clones the central repository:

```
In [30]: %%bash
         git clone file://$PWD/central.git alice
```

```
Cloning into 'alice'...
warning: You appear to have cloned an empty repository.
```

Alice enter her Git repository

```
In [31]: changedir('alice')
```

```
We are in directory /home/david/git-training/alice
```

0.3.2 Remote command

Without argument, *remote* command lists the **remote** repository Git know about:

```
In [32]: %%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 [33]: %%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) 00574cc] First line of foo.txt
 1 file changed, 1 insertion(+)
 create mode 100644 foo.txt
[master 67c105d] Second line of foo.txt
 1 file changed, 1 insertion(+)
```

0.3.3 Push command

Now, Alice wants to send her commits to the central repository, so that Bob can get them.
To send her commits to the central repository, Alice uses **push** command, whose arguments are:

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

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

```
In [34]: %%bash
         git push origin master:master
```

```
To file:///home/david/git-training/central.git
 * [new branch]      master -> master
```

```
In [35]: changedir(workdir)
```

```
We are in directory /home/david/git-training
```

Bob now clones the central repository too:

```
In [36]: %%bash
         git clone file://$PWD/central.git bob
```

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

```
In [37]: changedir('bob')
```

```
We are in directory /home/david/git-training/bob
```

It brings him the work of Alice.

```
In [38]: %%bash
         git log
```

```
commit 67c105d333ec9661af9caaa2065d937215559f76
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:18 2014 +0100

    Second line of foo.txt

commit 00574cc3f311eabc139ee19428a2ec9b64d66d90
Author: David Froger <david.froger@inria.fr>
Date:   Sun Nov 23 22:01:18 2014 +0100

    First line of foo.txt
```

Bob makes some changes, commits and pushes:

```
In [39]: %%bash
         echo "Third line" >> foo.txt
         git add foo.txt
         git commit -m "Third line to foo.txt"
         git push origin master:master
```

```
[master 6e72bc4] Third line to foo.txt
 1 file changed, 1 insertion(+)
```

```
To file:///home/david/git-training/central.git
   67c105d..6e72bc4  master -> master
```

0.3.4 Fetch command

```
In [40]: changedir('alice')
```

```
We are in directory /home/david/git-training/alice
```

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

```
In [41]: %%bash
         git fetch origin
```

```
From file:///home/david/git-training/central
   67c105d..6e72bc4  master     -> origin/master
```

0.3.5 Remote branches

Git **branch** command lists all branches of Alice's repository:

```
In [42]: %%bash
         git branch
```

```
* master
```

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

```
In [43]: %%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 on a remote. If the branch changes on the remote repository, use **fetch** again to refresh it.
To get all commits of `remotes/central/master.git` **remote branch** into **branch** master, merge it:

```
In [44]: %%bash
         git merge remotes/origin/master
```

```
Updating 67c105d..6e72bc4
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>
```

0.3.6 Pushing feature branch

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

```
In [45]: %%bash
         git branch exp
         git checkout 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 02b4b8f] First line of bar.txt
 1 file changed, 1 insertion(+)
 create mode 100644 bar.txt
[exp 9553996] Second line in bar.txt
 1 file changed, 1 insertion(+)

Switched to branch 'exp'
```

Alice then pushes her branch to the central repository, in a branch of the same name, which is created in the central repository.

```
In [46]: %%bash
         git push origin exp:exp
```

```
To file:///home/david/git-training/central.git
 * [new branch]      exp -> exp
```

0.3.7 Tracking branch

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

```
In [47]: changedir('bob')
```

```
We are in directory /home/david/git-training/bob
```

```
In [48]: %%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 a577df0] Third line in foo.txt
 1 file changed, 1 insertion(+)
[master 496cc62] Fourth line in foo.txt
 1 file changed, 1 insertion(+)
```

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

```
In [49]: %%bash
         git fetch
         git branch -a
```

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

From file:///home/david/git-training/central
 * [new branch]      exp        -> origin/exp
```

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

```
In [50]: %%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.
Tracking branch can also be used to use **pull** and **push** command without arguments.
Note that Alice and Bob can work on `exp` branch without disturbing `master` branch. - if `exp` was not a good idea, the branch can be dropped, - if `exp` is a good idea, it may be merged in master branch.

0.3.8 Visualizing branches

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

```
In [51]: %%bash
         echo "Third line" >> bar.txt
         git add bar.txt
         git commit -m "Third line in bar.txt"
```

```
[exp 90a0a76] Third line in bar.txt
 1 file changed, 1 insertion(+)
```

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

```
In [52]: %%bash
         git graph
```

```
* 90a0a76        (HEAD, exp) Third line in bar.txt (David Froger)
* 9553996        (origin/exp) Second line in bar.txt (David Froger)
* 02b4b8f        First line of bar.txt (David Froger)
| * 496cc62        (master) Fourth line in foo.txt (David Froger)
| * a577df0        Third line in foo.txt (David Froger)
|/
* 6e72bc4        (origin/master, origin/HEAD) Third line to foo.txt (David Froger)
* 67c105d        Second line of foo.txt (David Froger)
* 00574cc        First line of foo.txt (David Froger)
```

We note that: - Bob master branch has 2 more commits than the central one. - Bob master branch has 1 more commit that the central one.
The verbose option of **branch** is useful to see tracking branches:

```
In [53]: %%bash
         git branch -avv

* exp                    90a0a76 [origin/exp: ahead 1] Third line in bar.txt
  master                 496cc62 [origin/master: ahead 2] Fourth line in foo.txt
  remotes/origin/HEAD    -> origin/master
  remotes/origin/exp     9553996 Second line in bar.txt
  remotes/origin/master 6e72bc4 Third line to foo.txt
```

Bob pushes the commits of the two branches.

```
In [54]: %%bash
         git push origin exp:exp
         git push origin master:master

To file:///home/david/git-training/central.git
   9553996..90a0a76  exp -> exp
To file:///home/david/git-training/central.git
   6e72bc4..496cc62  master -> master
```

```
In [55]: %%bash
         git checkout master

         git push origin master:master
         git graph

Your branch is up-to-date with 'origin/master'.
* 90a0a76         (origin/exp, exp) Third line in bar.txt (David Froger)
* 9553996         Second line in bar.txt (David Froger)
* 02b4b8f         First line of bar.txt (David Froger)
| * 496cc62         (HEAD, origin/master, origin/HEAD, master) Fourth line in foo.txt (David Froger)
| * a577df0         Third line in foo.txt (David Froger)
|/
* 6e72bc4         Third line to foo.txt (David Froger)
* 67c105d         Second line of foo.txt (David Froger)
* 00574cc         First line of foo.txt (David Froger)

Switched to branch 'master'
Everything up-to-date
```

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

```
In [56]: %%bash
         # We already are in the master branch

         git merge exp
         git push origin master:master
         git branch -d exp
         git graph

Merge made by the 'recursive' strategy.
 bar.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 bar.txt
Deleted branch exp (was 90a0a76).
```

```
*   86e3524         (HEAD, origin/master, origin/HEAD, master) Merge branch 'exp' (David Froger)
|\
| * 90a0a76         (origin/exp) Third line in bar.txt (David Froger)
| * 9553996         Second line in bar.txt (David Froger)
| * 02b4b8f         First line of bar.txt (David Froger)
* | 496cc62         Fourth line in foo.txt (David Froger)
* | a577df0         Third line in foo.txt (David Froger)
|/
* 6e72bc4         Third line to foo.txt (David Froger)
* 67c105d         Second line of foo.txt (David Froger)
* 00574cc         First line of foo.txt (David Froger)

To file:///home/david/git-training/central.git
   496cc62..86e3524  master -> master
```

0.4 Distributed workflow

Committing to a central repository is ok for small team where developpers know each others

For large projects, it is blocking and dangerous to give write access to the central repository to an external contributor.

Git solves this problem with **forks**.

When a developper wants to contribute to a project, it forks the original bare repository. It means that the developper get its own bare repository. In this repository, he develops a feature in a branch. When the work his done, he asks an administrator to pull the feature branch of his repository to the master branch of the central repository.

This is called a **pull request**.

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

```
In [57]: changedir(workdir)
```

```
We are in directory /home/david/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 central repository.

```
In [58]: %%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 [59]: changedir('emma')
```

```
We are in directory /home/david/git-training/emma
```

Emma creates a topic branch, works on it, and pushes it:

```
In [60]: %%bash
         git branch baz
         git checkout 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 d205f1b] First line in baz.txt
 1 file changed, 1 insertion(+)
 create mode 100644 baz.txt
[baz e1388df] Second line in baz.txt
 1 file changed, 1 insertion(+)
```

```
Switched to branch 'baz'
To /home/david/git-training/central-emma-fork.git
 * [new branch]      baz -> baz
```

```
In [61]: changedir('alice')
```

```
We are in directory /home/david/git-training/alice
```

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

Note: Github provides functionnality to ask for pull request, 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 [62]: %%bash
         git remote add emma file://$HOME/git-training/central-emma-fork.git
         git fetch emma
```

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

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

```
In [63]: %%bash
         git checkout master
         git merge emma/baz
         git push origin master:master
```

```
Your branch is up-to-date with 'origin/master'.
Updating 6e72bc4..e1388df
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:///home/david/git-training/central.git
   86e3524..e1388df  master -> master
```