

# Version Control with git

R. Pastor-Satorras

Departament de Física  
UPC

Eines Informàtiques Avançades / EIA

# The ugly truth of scientific programming

- Scientific code is rarely a one-liner and not very often a very short piece of work: More often than not, it can be thousands of lines long
- Scientific code usually grows incrementally
  - ▶ We start coding some skeleton of program, and subsequently add different pieces of functionality
- We usually investigate at the time of coding
  - ▶ Have an idea, code it, check it, finally add it or eliminate it of the main code
- All this means that our code passes by several different versions, of different levels of precision/definition, that can produce different results depending on our checks

# The ugly truth of scientific programming

- This coding process is thus essentially a non-linear process, in which we can move forward, backwards and in loops, reusing and rewriting old pieces and implementing new ones
- Considering this, it is really very bad idea to work in a scientific code on a single source file, that we are adding, deleting, patching and amending in course of producing out final desired program
  - ▶ At some point of the development, how to recall a previous version of the code that was doing something we discarded before, but we want to use now?
- At a different level of programming difficulty, really complex programs (even scientific ones) are not the product of a single person, but the result of a collaboration between different scientists
- How can we handle this in a reasonable way?

# A lab notebook of computational physics

Another issue is related to *reproducible research*

- You start a big project, involving complex simulations, that might take months of years to complete, and involve several people
- After some time, you arrive to some results and create some figures presenting them
- You keep working on the project, finding new results
- After months, somebody requests how the figures were obtained, and maybe to perform some changes
- Unless you are able to keep track systematically of the evolution (i.e. different versions) of your code and results, you could be in big trouble

# Version control systems at the rescue

From computer science:

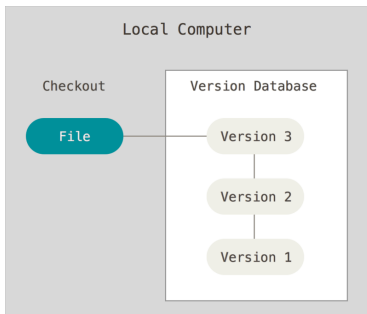
- Version control is a system than records changes performed to a file or set of files in time, and that allows to recall specific versions later

Version control systems permit:

- Revert files back to a previous state
- Revert an entire project back to a previous state
- Compare changes over time
- Know who and when last modified something that does not work
- Recover if you screw things up severely
- Work on different, parallel branches of the code, implementing and experimenting new ideas
- Manage work in the code by a team of people

# Local version control systems

## Poor-man's VCS

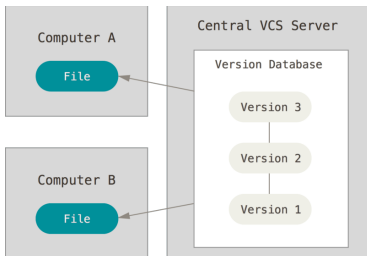


- Copy files into another time-stamped directory
- Allows to keep safe different versions of the code, separated from our working directory
- Error prone
  - ▶ Easy to forget in which directory you are, and mess with your saved versions
- Drawback: Everything is on the same disk, if it fails, all is lost

And how do you collaborate in this framework?

# Centralized version control systems

Specially to favor collaboration, but can be used stand-alone

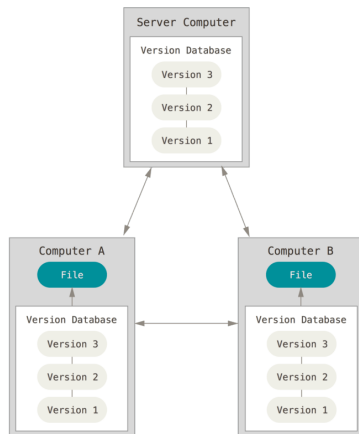


- A single server contains all versioned files
- A number of clients check out files from the server
- The files are updated and later committed back to the server
- Drawback: the single point of failure that the centralized server represents
- If the disk of the server becomes corrupted, all is lost

# Distributed version control systems

Latest, most popular VCS technology

- Clients don't just check out the latest version of files, they fully mirror the repository
- Every client is a clone of the server repository; if any server dies, any of the client repositories can be copied back up to the server to restore it
- Clients can work in parallel on different parts of the code, and commit the changes to the central server when they are done





# The git distributed version control systems

git is one of the most popular distributed version control systems.

Developed by Linus Torvalds in need of a VCS capable for the development of the Linux kernel

git is extremely powerful

- Ex. code base of Linux kernel: 9 million lines of code distributed over 25.000 files, subject by all kinds of complex manipulations by hundreds of contributors.

However ...

Git is a saw with no guard that makes it easy to cut your own arm off.  
But it also comes with an easy arm reattachment kit.  
And you can even attach the arm to your knee if you want.

—Wayne Conrad

We have to work carefully with git

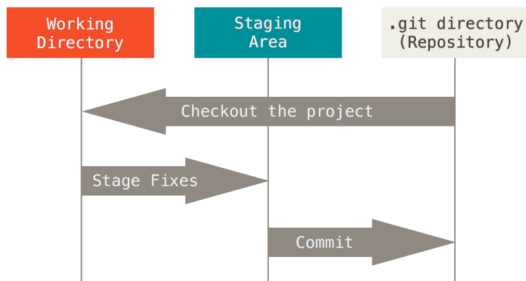
# How git works

- git stores complete snapshots of a project, identified by a *repository*, i.e. a folder
- Operations consist essentially in adding data to create and update the different snapshots. Since there are no deletions, it is very difficult to destroy your work
- git performs a check-sum of the data, so that is impossible to change the contents of the folder without git knowing it
- git identifies many things by a SHA-1 hash string, of the form

24b9da6552252987aa493b52f8696cd6d3b00373

We refer to these strings (just the first few characters) to identify the different changes and snapshots

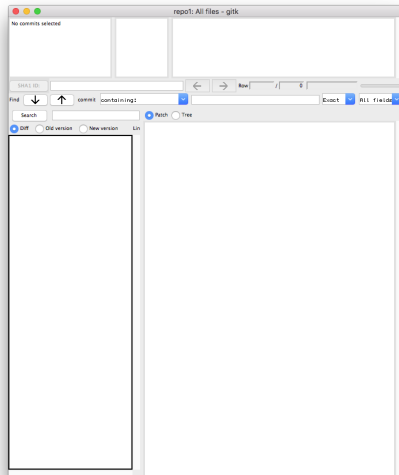
# The three states of git



- Define a folder as a repository
- Modify files on the folder (working directory)
- *Stage* files, as soon as you are done working with them, on the *staging area*. They are not yet saved in this area, just ready to be saved.
- Do a *commit*, taking the files in the staging area and permanently storing a snapshot of the project onto the repository

# Local version control with git

- Start version controlling in our computer, to keep track of the work inside a working directory
- We will use git on the command line
- You can also use GUI versions (ex. gitk)



# Installing git

- Go to the Git website
- Follow the instructions for your platform.

You have it installed in the computers in the room. Open a terminal (preferably in Linux) and get ready to `git`...

# The git command

```
> git --help
usage: git [--version] [--help] [-C <path>] [-c name=value]
        [--exec-path=<path>] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        <command> [<args>]
```

These are common Git commands used in various situations:

start a working area (see also: `git help tutorial`)

clone	Clone a repository into a new directory
init	Create an empty Git repository or reinitialize an existing one

work on the current change (see also: `git help everyday`)

add	Add file contents to the index
mv	Move or rename a file, a directory, or a symlink
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index

examine the `history` and state (see also: `git help revisions`)

bisect	Use binary search to find the commit that introduced a bug
grep	Print lines matching a pattern
log	Show commit logs
show	Show various types of objects
status	Show the working tree status

grow, mark and tweak your common `history`

branch	List, create, or delete branches
checkout	Switch branches or restore working tree files
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
merge	Join two or more development histories together

# Initializing git

First of all, we can provide some initial information about ourselves to git:

```
> git config --global user.name "John Doe"
> git config --global user.email johndoe@example.com
> git config --global core.editor vim
```

git will use the information to tag our commits.

Checking the status

```
> git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

# Creating a repository

```
> mkdir project
> cd project/
:project> git init
Initialized empty Git repository in project/.git/
:project> git status
On branch master
```

Initial commit

nothing to commit (create/copy files and use "git add" to track)



# Inside the repository

```
:project> ls -alg
total 0
drwxr-xr-x  3 staff 102 Mar 29 13:03 .
drwxr-xr-x  8 staff 272 Mar 29 13:03 ..
drwxr-xr-x 10 staff 340 Mar 29 13:05 .git
```

# Inside the repository

```
:project> ls -alg
total 0
drwxr-xr-x  3 staff 102 Mar 29 13:03 .
drwxr-xr-x  8 staff 272 Mar 29 13:03 ..
drwxr-xr-x 10 staff 340 Mar 29 13:05 .git

:project> ls -alg .git/
total 12
drwxr-xr-x 10 staff 340 Mar 29 13:05 .
drwxr-xr-x  3 staff 102 Mar 29 13:03 ..
-rw-r--r--  1 staff  23 Mar 29 13:03 HEAD
drwxr-xr-x  2 staff  68 Mar 29 13:03 branches
-rw-r--r--  1 staff 137 Mar 29 13:03 config
-rw-r--r--  1 staff  73 Mar 29 13:03 description
drwxr-xr-x 11 staff 374 Mar 29 13:03 hooks
drwxr-xr-x  3 staff 102 Mar 29 13:03 info
drwxr-xr-x  4 staff 136 Mar 29 13:03 objects
drwxr-xr-x  4 staff 136 Mar 29 13:03 refs
```

# Staging files

```
:project> vim hello.f08      # Adding a file
:project> cat hello.f08
program hello
    implicit none

    print *, "Hello World!"
end program hello

:project> git add hello.f08    # staging a file
```

```
:project> git status      # checking the status
On branch master
```

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: hello.f08

Untracked files:

(use "git add <file>..." to include in what will be committed)

hello.f08~

# Ignoring files

Text editors usually create back-up files (the `*.*~` file in the present case), which we do not to keep track of

We can specify files to ignore in a `.gitignore` file

```
:project> vim .gitignore      # Edit .gitignore
:project> cat .gitignore
*~
.*
```

```
:project> git status      # check status again
On branch master
```

Initial commit

Changes to be committed:

(use `"git rm --cached <file>..."` to unstage)

new file: hello.f08

# Saving (committing) a snapshot

*# committing staged files*

```
:project> git commit -m "Initial file"
[master (root-commit) 8715df9] Initial file
 1 file changed, 5 insertions(+)
 create mode 100644 hello.f08
```

*# checking*

```
:project> git status
On branch master
nothing to commit, working directory clean
```

The git commit command

- Saves the snapshot, officially called a “revision”
- Gives that snapshot a unique ID number (a revision hash), here 8715df9, which we use to identify it
- Names you as the author of the changes
- Allows you, the author, to add a message [-m “Initial file”]

# Viewing the history

```
:project> git log  
commit 8715df975a4770faf7699e7348679c19c53e8e4d  
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>  
Date: Tue Mar 29 13:36:45 2016 +0200
```

Initial file

The git log command

- Prints the logged metadata for each commit
- Each commit possesses a unique (hashed) identification number that can be used to refer to that commit (8715df9, the first numbers are enough)
- Records the date and time at which the commit occurred.
- The log message for each commit is printed along with that commit.

# Making and committing changes

*# edit the file*

```
:project> vim hello.f08
```

```
:project> cat hello.f08
```

```
program hello
  implicit none
  character(len=50) :: name

  print *, "Enter your name"
  read *, name

  print *, "Hello ", name, "!"
end program hello
```

*# checking status*

```
:project> 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:   hello.f08
```

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

# Making and committing changes

```
# stage for the snapshot
:project> git add hello.f08
# commit
:project> git commit -m "Improved program with name"
[master 348dcf7] Improved program with name
1 file changed, 5 insertions(+), 1 deletion(-)
```



# What have we done?

```
# check status
```

```
:project> git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

```
:project> git log
```

```
commit 348dcf72120cdcf7cfcfdc203d55ff8b3fbff917
```

```
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
```

```
Date: Tue Mar 29 13:53:44 2016 +0200
```

```
Improved program with name
```

```
commit 8715df975a4770faf7699e7348679c19c53e8e4d
```

```
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
```

```
Date: Tue Mar 29 13:36:45 2016 +0200
```

```
Initial file
```

# Making more commits

```
:project> vim hello.f08
:project> cat hello.f08
program hello
    implicit none
    character(len=50) :: name, surname

    print *, "Enter your name"
    read *, name
    print *, "Enter your surname"
    read *, surname

    print *, "Hello ", name, surname, "!"
end program hello

:project> git add hello.f08

:project> git commit -m "More improvements"
[master 904f5dd] More improvements
 1 file changed, 4 insertions(+), 2 deletions(-)
```

- Now we have three snapshots, each corresponding to the three commits (hash numbers) made

# Looking at the differences

```
# status of the repository, with differences in snapshots
:project> git log -p
commit 904f5dd494de8d6703b3764eb96fd71665a9d564
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date:   Tue Mar 29 14:35:40 2016 +0200
```

More improvements

```
diff --git a/hello.f08 b/hello.f08
index df0e53a..4609e87 100644
--- a/hello.f08
+++ b/hello.f08
@@ -1,9 +1,11 @@
 program hello
   implicit none
-  character(len=50) :: name
+  character(len=50) :: name, surname

   print *, "Enter your name"
   read *, name
+  print *, "Enter your surname"
+  read *, surname

-  print *, "Hello ", name, "!"
+  print *, "Hello ", name, surname, "!"
 end program hello

commit 348dcf72120cdcf7cfcfdc203d55ff8b3fbff917
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date:   Tue Mar 29 13:53:44 2016 +0200
:
```

# Reverting to a previous snapshot

```
# log in one line format
:project> git log --pretty=oneline
904f5dd494de8d6703b3764eb96fd71665a9d564 More improvements
348dcf72120cdcf7cfcfdc203d55ff8b3fbff917 Improved program with name
8715df975a4770faf7699e7348679c19c53e8e4d Initial file

# present file
:project> cat hello.f08
program hello
  implicit none
  character(len=50) :: name, surname

  print *, "Enter your name"
  read *, name
  print *, "Enter your surname"
  read *, surname

  print *, "Hello ", name, surname, "!"
end program hello

# bring the first version, identified by the first 7 digits of the hash
:project> git reset --hard 8715df
HEAD is now at 8715df9 Initial file
:project> cat hello.f08
program hello
  implicit none

  print *, "Hello World!"
end program hello

# bring again the last version
:project> git reset --hard 904f5dd
HEAD is now at 904f5dd More improvements
```

# Branches in git

Branches are parallel instances of a repository that can be edited and version controlled in parallel.

They are useful for pursuing various implementations experimentally or maintaining a stable core while developing separate sections of a code base.

List the branches in a repository

*# Listing branches*

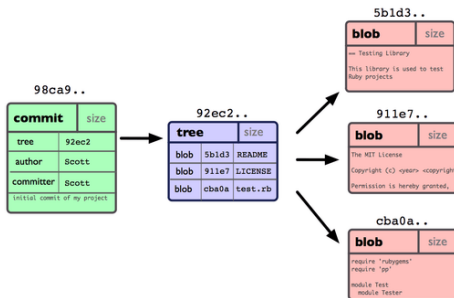
```
:project> git branch  
* master
```

The “master” branch is created when the repository is initialized and the first commit made.

This is the default branch and is conventionally used to store a clean master version of the source code.

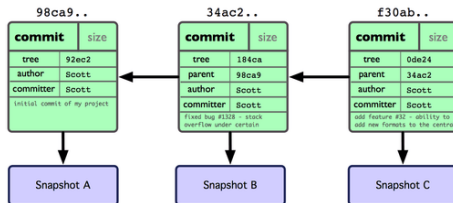
# Summary: how git works

A Git repository with an initial commit contains objects: one blob for the contents of each of your files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with a pointer to that root tree and all the commit metadata.



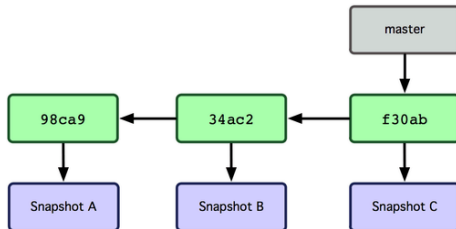
# Summary: how git works

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it. After two more commits, your history might look something like this



## Summary: how git works

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master`. As you initially make commits, you're given a `master` branch that points to the last commit you made. Every time you commit, it moves forward automatically.

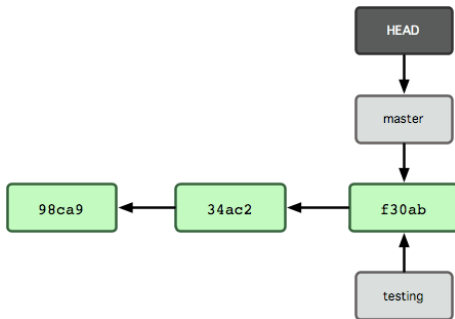




# Branches in git

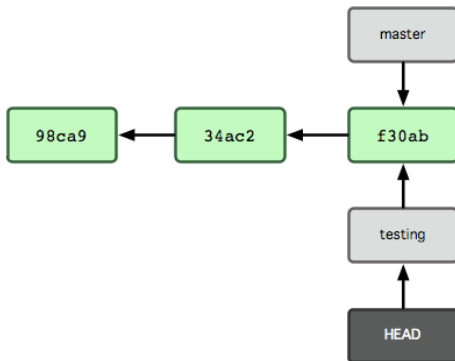
When you create a new branch, a new pointer is created for you to move around.

Git knows in which branch you are by keeping a special pointer called HEAD, pointing to the local branch you're currently on



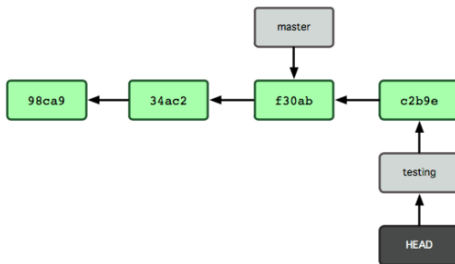
# Branches in git

You can move to the newly created branch. HEAD will then point to it. Now you can start performing changes to your code



# Branches in git

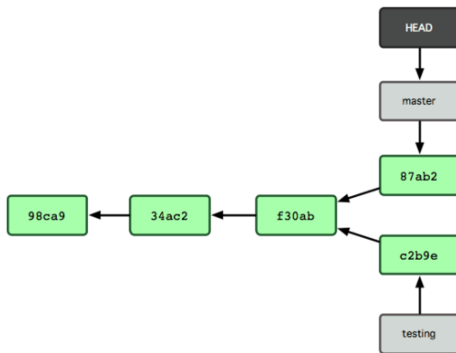
You can start making commits in the new branch. The HEAD in the branch moves forward, but your master branch still points to the commit you were on when you changed to the branch



# Branches in git

You can switch back to `master`, perform some more changes and commits. All changes take place in isolated branches, and it is possible to go back and forth among them.

When you are done with the new branch you can delete it, keep it as a separate code base, or merge it with `master`



# Creating a branch

*# Creating a new branch*

```
:project> git branch experiment
```

*# Listing branches*

```
:project> git branch
  experiment
* master
```

We have now two branches, master and experiment. The star \* indicates the branch in which we are

To switch to a branch

*# Switch to branch*

```
:project> git checkout experiment
```

Switched to branch 'experiment'

*# list*

```
:project> git branch
* experiment
  master
```

# Working on a branch

```
# Add a file to the branch
:project> vim hello.c
:project> cat hello.c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    fprintf(stderr, "%s\n", "Hello world!");

    exit(0);
}
:project> git status
On branch experiment
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    hello.c

nothing added to commit but untracked files present (use "git add" to track)
:project> git add hello.c
:project> git commit -m "Added C version"
[experiment 61704b3] Added C version
 1 file changed, 10 insertions(+)
 create mode 100644 hello.c
```

We can add stuff to the branch, developing some new idea

# Switching branches

```
# check where we are
:project> git branch
* experiment
  master
:project> ls -l      # list files in the repository
total 8
-rw-r--r-- 1 romu staff 113 Mar 29 15:05 hello.c
-rw-r--r-- 1 romu staff  0 Mar 29 15:04 hello.c~
-rw-r--r-- 1 romu staff 222 Mar 29 14:50 hello.f08
-rw-r--r-- 1 romu staff  0 Mar 29 13:12 hello.f08~
# back to master branch
:project> git checkout master
Switched to branch 'master'
:project> git branch
  experiment
* master
:project> ls -l      # list files in the repository
total 4
-rw-r--r-- 1 romu staff  0 Mar 29 15:04 hello.c~
-rw-r--r-- 1 romu staff 222 Mar 29 14:50 hello.f08
-rw-r--r-- 1 romu staff  0 Mar 29 13:12 hello.f08~
```

Back in master, observe that the file "hello.c" has disappeared (it's in the experiment branch!). Only the back-up file from the text editor remains on the working directory

# Merging branches

Once the experiment is finished, if we are happy with it, we can incorporate it to the master branch

```
# go to the master branch
:project> git checkout master
Already on 'master'
# merge the branch experiment to the master
:project> git merge experiment
Updating 904f5dd..61704b3
Fast-forward
 hello.c | 10 ++++++++
 1 file changed, 10 insertions(+)
 create mode 100644 hello.c
# now "hello.c" is in master
:project> ls -l
total 8
-rw-r--r-- 1 romu staff 113 Mar 29 15:15 hello.c
-rw-r--r-- 1 romu staff  0 Mar 29 15:04 hello.c~
-rw-r--r-- 1 romu staff 222 Mar 29 14:50 hello.f08
-rw-r--r-- 1 romu staff  0 Mar 29 13:12 hello.f08~
# delete the now useless branch
:project> git branch -d experiment
Deleted branch experiment (was 61704b3).
```



# Merging branches

```
# final status
:project> git branch
* master
:project> git log
commit 61704b3789a8426e504259b70351e22a5f6d7df8
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date: Tue Mar 29 15:08:15 2016 +0200
```

Added C version

```
commit 904f5dd494de8d6703b3764eb96fd71665a9d564
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date: Tue Mar 29 14:35:40 2016 +0200
```

More improvements

```
commit 348dcf72120cdcf7cfcfdc203d55ff8b3fbff917
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date: Tue Mar 29 13:53:44 2016 +0200
```

Improved program with name

```
commit 8715df975a4770faf7699e7348679c19c53e8e4d
Author: Romualdo Pastor Satorras <romualdo.pastor@gmail.com>
Date: Tue Mar 29 13:36:45 2016 +0200
```

Initial file

The last commit has been imported from the branch

# The power of branches

- 1 Branches in git allow to make temporal explorations of some coding project, developing different ideas in parallel to explore their potential
- 2 Once we are satisfied with one or several of them, we can merge them into the master code repository
- 3 Branches allow also for team coding: Several aspects of the same codebase can be worked by different people using parallel branches, which are finally merged when the different pieces are ready

## Merging conflicts

If two branches contain files with are different, they cannot be merged directly, and the merge has to be aborted

# Remote version control with git

The full power of git emerges when it is combined with an online repository hosting system, which allows

- 1 Back up online automatically your code (thus making a security copy outside your computer)
- 2 Managing files in a collaboration
- 3 Forking remote repositories to modify or extend code of other people

# Repository hosting

There are many repository hosting services online, that serve to store and access repositories

- Launchpad
- Bitbucket
- Google Code
- SourceForge
- GitHub
- GitLab

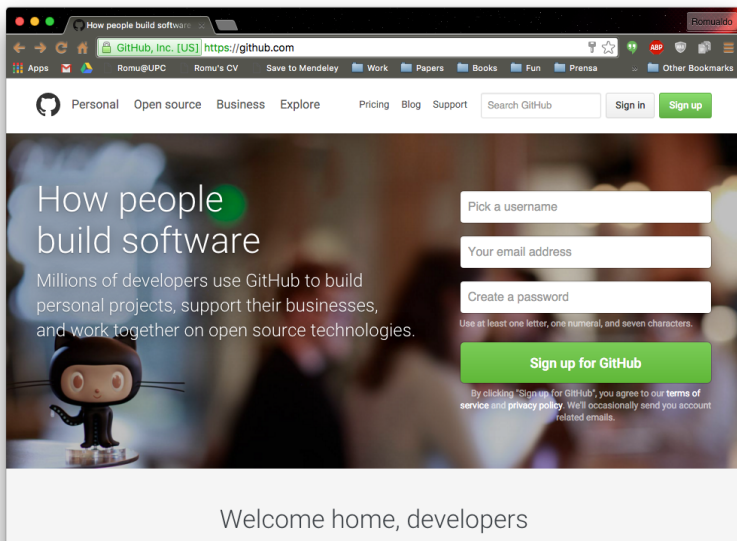
We will use here GitHub, because it is very easy to work in it, providing many tools for browsing, documenting, collaborating, etc

It can be used free, but it has the restriction that all repositories are public (anybody can look at them)

If you want free privacy, go for GitLab

# Create a free account on GitHub

Take a moment to create a free account



The screenshot shows a web browser window with the GitHub homepage. The browser's address bar shows "https://github.com". The page features a navigation bar with links for Personal, Open source, Business, Explore, Pricing, Blog, and Support. A search bar and "Sign in" and "Sign up" buttons are also present. The main content area has a large heading "How people build software" and a subheading "Millions of developers use GitHub to build personal projects, support their businesses, and work together on open source technologies." Below this is a small illustration of the GitHub mascot, Octocat. To the right of the text are three input fields for "Pick a username", "Your email address", and "Create a password". Below the password field is a note: "Use at least one letter, one numeral, and seven characters." A large green "Sign up for GitHub" button is positioned below the input fields. At the bottom of the sign-up section, there is a small disclaimer: "By clicking 'Sign up for GitHub', you agree to our terms of service and privacy policy. We'll occasionally send you account related emails." The footer of the page says "Welcome home, developers".

How people build software

Millions of developers use GitHub to build personal projects, support their businesses, and work together on open source technologies.

Pick a username

Your email address

Create a password

Use at least one letter, one numeral, and seven characters.

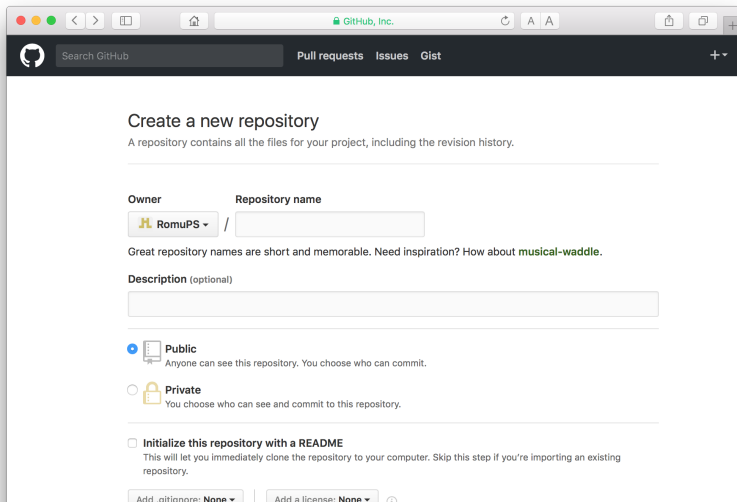
**Sign up for GitHub**

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We'll occasionally send you account related emails.

Welcome home, developers

# Creating a repository on GitHub

Go to the main page in your GitHub account and create a new project, clicking in the corresponding big green button. Give it the name "project"



The screenshot shows the GitHub web interface for creating a new repository. The browser's address bar shows 'GitHub, Inc.'. The GitHub logo and navigation links are at the top. The main heading is 'Create a new repository' with a subtitle 'A repository contains all the files for your project, including the revision history.' Below this, there are three main sections: 'Owner' with a dropdown menu showing 'RomuPS', 'Repository name' with an empty text box, and 'Description (optional)' with an empty text box. There are two radio button options for visibility: 'Public' (selected) and 'Private'. At the bottom, there is an unchecked checkbox for 'Initialize this repository with a README'. At the very bottom, there are dropdown menus for 'Add a file with: None' and 'Add a license: None'.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: RomuPS / Repository name:

Great repository names are short and memorable. Need inspiration? How about **musical-waddle**.

Description (optional):

☒ **Public**  
Anyone can see this repository. You choose who can commit.

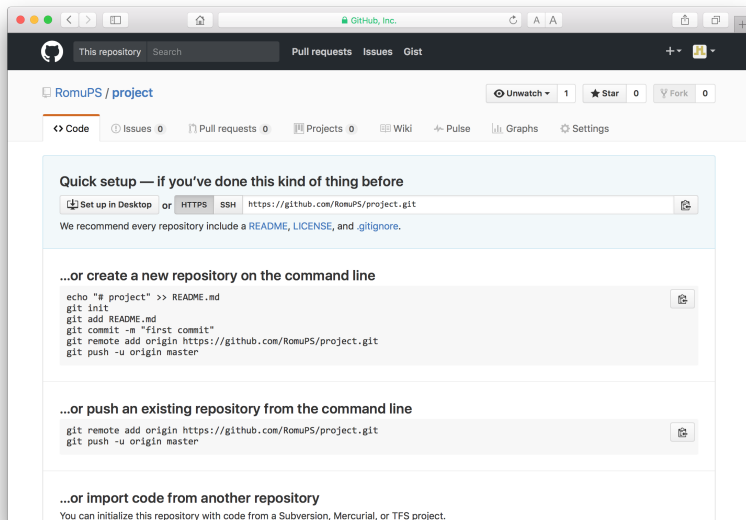
☐ **Private**  
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add a file with: None | Add a license: None

# Creating a repository on GitHub

The repository is now empty, and GitHub proposed several ways to fill it



# Declaring a remote

To synchronize changes between a local repository and a remote repository, the location of the remote must be registered on the local repository

```
# declare remote
```

```
:project> git remote add origin https://github.com/RomuPS/project.git
```

- remote makes git register something about a remote repository
- add declares the alias of the remote
- origin is the conventional alias for the repository
- URL is copied from the GitHub page

```
# check remotes
```

```
:project> git remote -v
```

```
origin          https://github.com/RomuPS/project.git (fetch)
```

```
origin          https://github.com/RomuPS/project.git (push)
```



# Sending commits (push)

The git push command sends (pushes) commits from a branch of the local repository into the remote repository

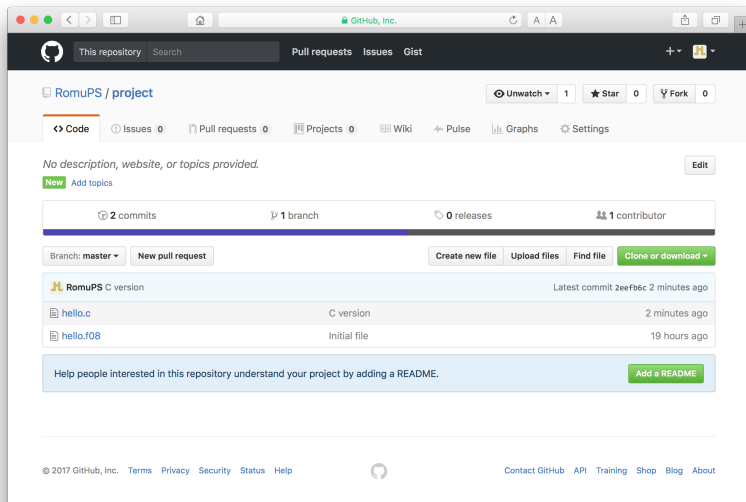
*# push commits*

```
:project> git push origin master
Username for 'https://github.com': RomuPS
Password for 'https://RomuPS@github.com':
Counting objects: 12, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (12/12), 1.19 KiB | 0 bytes/s, done.
Total 12 (delta 2), reused 0 (delta 0)
To https://github.com/RomuPS/project.git
* [new branch]      master -> master
```

- Push "master" into "origin"
- Requires GitHub username and password

# Sending commits (push)

Now our GitHub project is populated with the files from the local repository



# Sending commits (push)

Now we can make more commits in the local repository and push them on the remote using the same steps

```
# Add a new file
:project> vim hello.py
:project> cat hello.py
print("Hello World!")
:project> git add hello.py
# commit the changes
:project> git commit -m "Added python version"
[master 15001ca] Added python version
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
# push the changes
:project> git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 339 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/RomuPS/project.git
 61704b3..15001ca  master -> master
```

The changes have been pushed to the remote GitHub (check!)

# Pushing branches

```
:project> git branch experiment
:project> git branch
      experiment
* master
:project> git checkout experiment
Switched to branch 'experiment'
:project> vim hello.jl
:project> git add hello.jl
:project> git commit -m "Added Julia version"
:project> git push origin master
Everything up-to-date
# oops! We must push the branch explicitly
:project> git push origin experiment
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 366 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/RomuPS/project.git
 * [new branch]      experiment -> experiment
```

We can check in the GitHub page that now two branches appear, "master" and "experiment"

# Cloning a Github repository

Github allows to *clone* an entire repository of another person into our local machine (that is, if the repository is public)

We will have the complete repository, with all its contents, at our disposition, to edit and play with it

Example: Let us clone a nice repository of emacs configurations for people trying to escape from vim: **Emacs for the stubborn martian vimmer**

<https://github.com/hlissner/doom-emacs>,

```
> git clone https://github.com/hlissner/doom-emacs ~/.emacs.d
Cloning into ' ~/.emacs.d'...
remote: Counting objects: 13802, done.
remote: Compressing objects: 100% (115/115), done.
remote: Total 13802 (delta 64), reused 0 (delta 0), pack-reused 13685
Receiving objects: 100% (13802/13802), 4.31 MiB | 1.05 MiB/s, done.
Resolving deltas: 100% (9811/9811), done.
```

The contents is in our emacs configuration folder `.emacs.d`

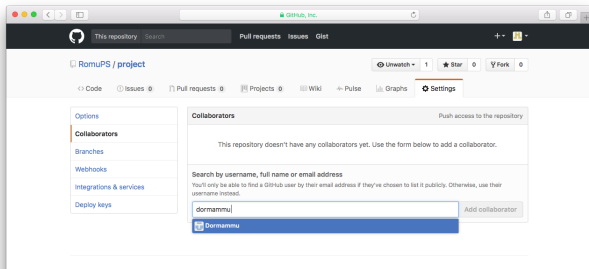
We can now start messing and playing with it.

But... we don't have permissions to push anything into the github repository

# Collaborating with GitHub

GitHub allows the collaboration of different users on the same repository  
There are two possibilities for collaboration:

- In the tab Settings, define a collaborator, by searching his/her username or email address. This gives the collaborator(s) full access to push data to the remote repository
- While this is simple, it can lead to problems
  - ▶ Two people trying to push the same file, with different versions
  - ▶ Can lead to conflicts, which are difficult to observe and fix, since everybody has the same privileges



# Cloning and forking

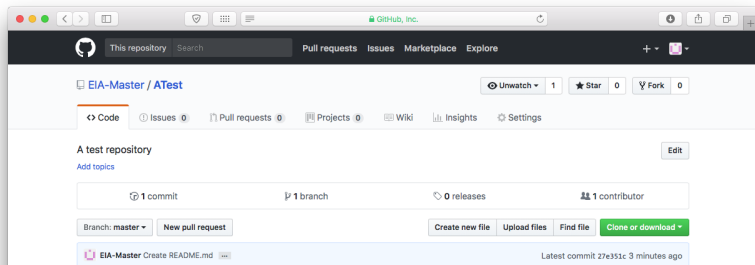
Another approach to collaborating on GitHub consists in **forking** an already existing project

To start the collaboration, some manager creates a main repository, where the main version of the code base will be maintained

Collaborators, on their GitHub page, locate the page of the manager, and the main repository.

On this page, collaborators click on the button Fork

This creates in your own GitHub account a full copy of the main repository



# Cloning a remote repository

To have access to the forked repository we just created, we have to clone it, from our GitHub space, into our local machine (user your own github username!!)

```
> git clone https://github.com/RomuPS/ATest.git
Cloning into 'ATest'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
> cd ATest/
:ATest> ls -l
total 4
-rw-r--r-- 1 romu staff 30 Feb 23 16:06 README.md
:ATest> git remote -v
origin      https://github.com/RomuPS/ATest.git (fetch)
origin      https://github.com/RomuPS/ATest.git (push)
```

The cloning operation directly links our local to our remote copy of the main repository, with the alias "origin"

In this way, we can start making changes and updating our own version of the main repository



# Updating a cloned remote repository

But in the mean time, the main repository can have been changed.

To update the changes in the main repository into our local version, we have to link it to the main, for which we choose the alias "main"

```
:ATest> git remote add main https://github.com/EIA-Master/ATest.git
:ATest> git remote -v
main          https://github.com/EIA-Master/ATest.git (fetch)
main          https://github.com/EIA-Master/ATest.git (push)
origin        https://github.com/RomuPS/ATest.git (fetch)
origin        https://github.com/RomuPS/ATest.git (push)
```

Now we can fetch changes in main

```
:ATest> git fetch main
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
....
```

```
:ATest> git pull main master
From https://github.com/EIA-Master/ATest
 * branch          master      -> FETCH_HEAD
Updating dd2836b..911b3d5
....
```

# Working in a cloned remote repository

We now work on our cloned repository, adding a file

```
:ATest> vim stuff.txt
:ATest> cat README.md
A file with some stuff

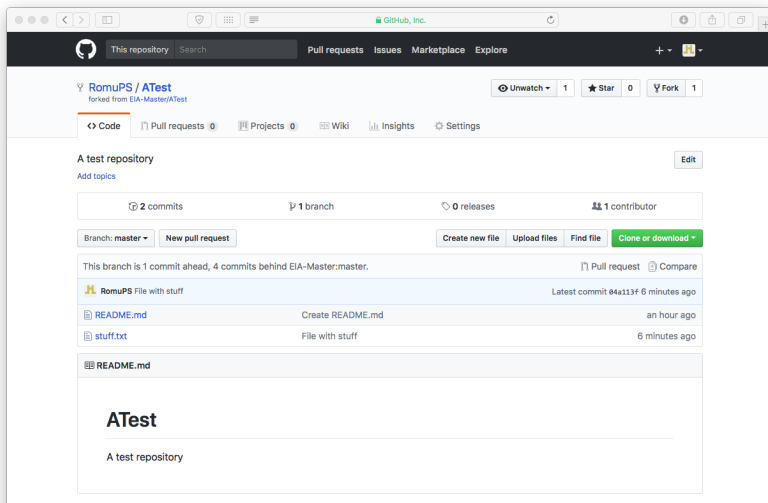
:ATest> git add stuff.txt
:ATest> git commit -m "File with stuff"
[master 04a113f] File with stuff
 1 file changed, 2 insertions(+)
 create mode 100644 stuff.txt
# push to our cloned remote "origin"
:ATest> git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/RomuPS/ATest.git
 27e351c..04a113f  master -> master
```

But since we don't know the password for the main repository "main", we cannot directly push there our changes

```
:ATest> git push main master
remote: Permission to EIA-Master/ATest.git denied to RomuPS.
fatal: unable to access 'https://github.com/EIA-Master/ATest.git/':
The requested URL returned error: 403
```

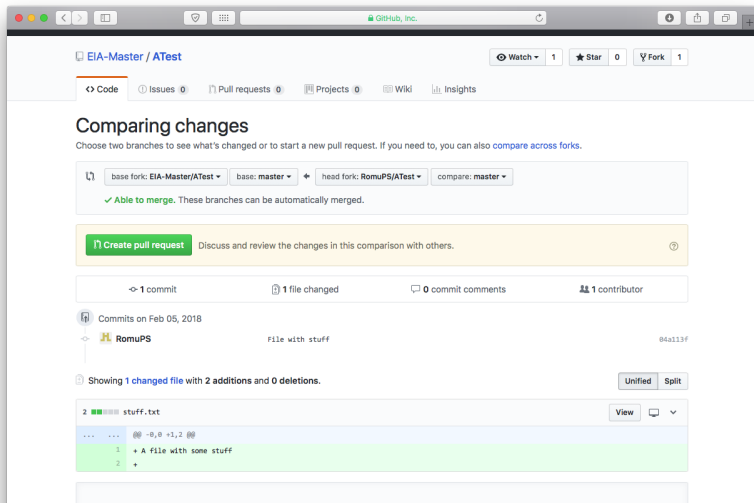
# Pulling to the main remote repository

To pull changes to the main, we have to open a "New pull request" in our GitHub web page



# Pulling to the main remote repository

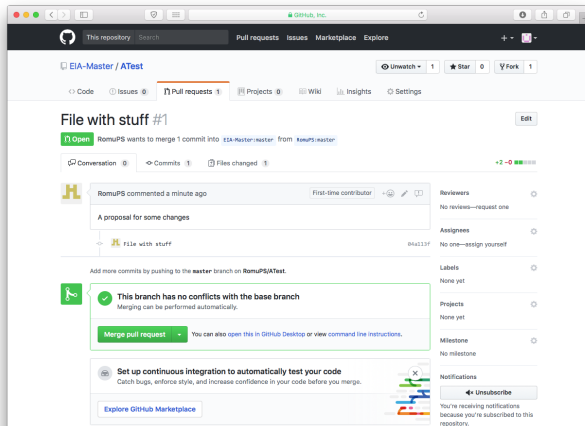
We have to choose the base fork (the main) and our fork, choose the branches we want to pull, add a comment, and submit



# Pulling to the main remote repository

In the GitHub page of the owner of main, appears a pull request, in the corresponding tab. The owner of main can check it, see if there are no conflicts, and accept (merge) it.

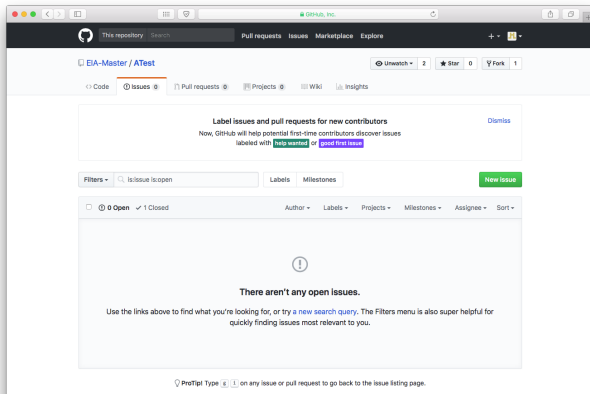
Both remotes (origin and main) will be now synchronized



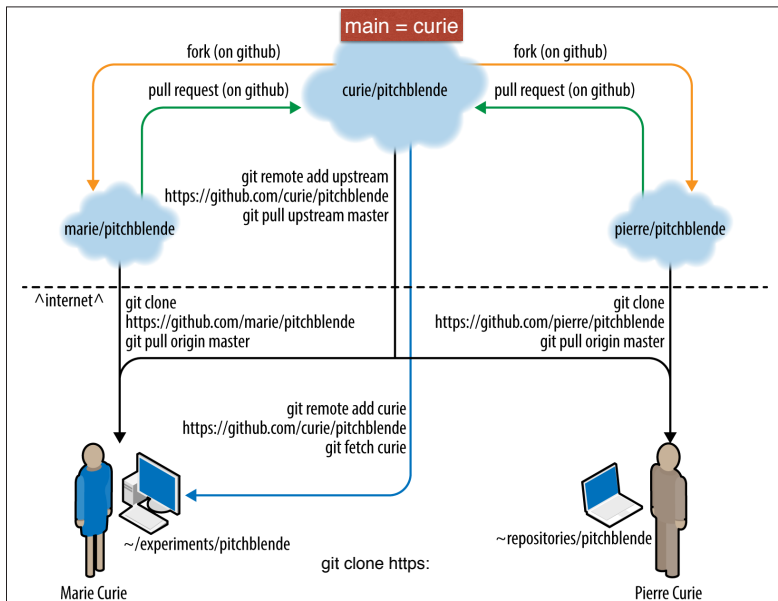
# Managing collaboration

Apart from adding new pieces of code, GitHub allows discussion on problems (issues) in the codebase. To start a discussion, we have to open a new issue in the Issues tab of the main repository.

The discussion can be continued by the members of the collaboration, and closed when the problem raised has been solved.



# Forking and cloning sum up



# Collaboration exercise

We are going to develop a fake project in which I will play the role of boss.

- Go to my github page (RomuPS user) and fork and clone the repository `Project_mock_up`
- Link your own copy to my folder as main
- Open a branch of the project in your own local folder, add some source code for something simple and merge it into your master
- Make a pull request into my account
- Let us see what happens ...