Git{Hub,Lab} -- *“everything is local”*

–= A beginner’s guide =–

Last update: July 2024

Author: Cedric Bhihe

Corrections and suggestions are welcome and should be consigned in the *Issues* section of the https://www.github.com/Cbhihe/quickstart_git repo.

Foreword

Writing this document began as an expanded version of another quick-start document, authored by Jordi Torres and available on Github. It has undergone several full rewrites and has grown to become an almost 30 page long manual in the form of a step by step tutorial.

As the original document put together by Jordi Torres, it is meant as a hands-on introduction to Git{Hub,Lab} for beginners. From scratch, it should get you started and operational under two hours. It deals with the most salient (and basic) aspects of how to get on and about with git version control. If you would rather have a lengthier, more detailed description of Git, go to *Pro git* by Scott Chacon and Ben Straub, a freely available 514 page long digital reference. Note that this Quickstart tutorial was not written with *Pro git* in hand and is in no way a summary skimmed off that book. In fact I have not read *Pro git* (yet) even though I have had a look at it on rare occasions.

Corrections and suggestions are welcome and should be consigned in the Issues section of this repo.

Copyright Notice

Table of Contents

Foreword.....1
Copyright Notice.....1
1. Getting Started.....3
2. Getting started with Git.....4
2.1. Installing Git.....4
2.2. Initialize a local Git repository: git init.....4
2.3. Add & commit.....5
2.4 Files' status and stage.....7
2.5 Removing already staged or committed files.....7
2.5.1 – Remove from staging area.....7
2.5.2 – Remove single file from committed area.....7
2.5.3 – Correct mistakes on commits and / or pushes.....7
2.6. Branching.....9
2.7. Remote branches: fetch, merge and pull.....10
2.8. Renaming branches.....13
2.8.1 – On the remote repo (at "origin").....13
2.8.2 – On the locally cloned repo.....13
2.9. Renaming repos.....13
2.9.1 – Renaming the remote repo ("origin").....13
2.9.2 – Renaming the local repo.....14
2.10. Removing / deleting branches.....14
2.10.1 – Deleting a local branch.....14
2.10.2 – Deleting a remote branch.....14
2.11. Fetching remote branches.....15
3. Getting started with GitHub.....15
3.1. Creating a GitHub account.....15
3.2. Connecting a local repo to a Github repository.....15
3.3. Branching.....17
3.4. Exclude files when pushing to a remote repo with "./.gitignore".....18
3.4.2 – How to subject an already staged or committed file to exclusion directives.....19
3..4.3 – Creating .gitignore exclusion patterns.....19
3.5 Inviting collaborators to a project.....21
3.6. Securing an SSH connection to your GitHub account.....21
3.6.1 – Creating an SSH key.....21
3.6.2 – Adding an SSH key to the SSH-agent.....22
3.6.3 – Adding an SSH key to a GitHub account.....22
3.6.4 – Test the SSH connection.....22
3.6.5 – Troubleshooting.....23
4. Git clone.....23
5. To go into more detail.....24
Appendix A.....25
Markdown format.....25
Appendix B.....26

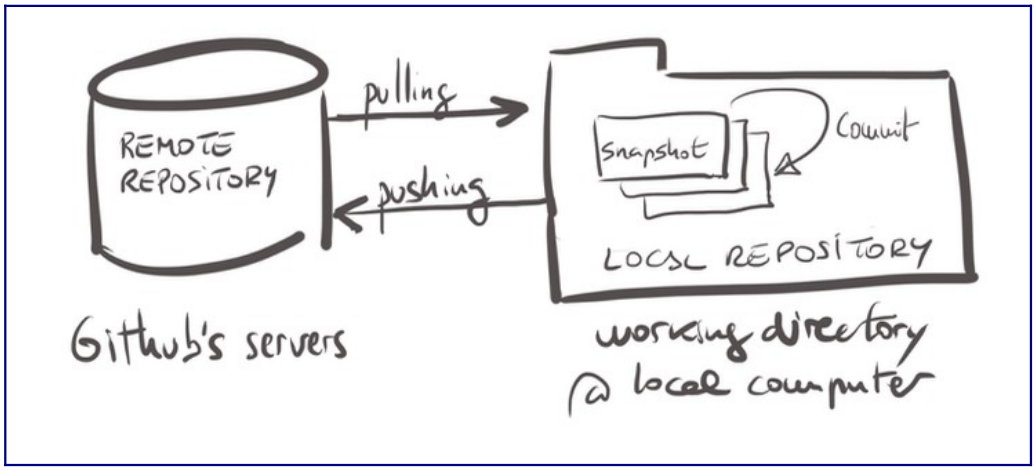
This hands-on tutorial is a beginner's guide to one of the best known distributed version-control system (dVCS): Git. It provides a step-by-step introduction to its basic characteristics. It is intended for an audience with either no prior knowledge, or a need for a refresher due to insufficient practice. Its main objective is to bring all to the required knowledge level necessary to use Git at a passing level. This document was originally written and later edited and expanded based on the belief that hands-on practice is paramount when acquiring new technical knowledge. To benefit from this tutorial, follow instructions on a computer running Linux and connected to the internet. You should have **`sudo`** privileges to be able to install packages on your local host.

1. Getting Started

Git is a free and open-source dVCS, launched in 2005 by Linus Torvald. It is designed to handle everything from small to very large code projects. With Git, **users keep entire code files on their location machines**. Git also keeps a historical record of the files in your codebase. It manage workflows, in such a way that teams of programmers may work together and revert any changes by rolling back their code to a previous version. To achieve that Git stores files into a repository typically hosted on GitHub (**G+**) or on GitLab.

Clarifying some vocabulary and key concepts:

- **Snapshot** is the way Git keeps track of code history, recording what files look like at a given point in time. We can decide when to take a snapshot and go back to visit any previous snapshot.
- **Commit** refers to the act of creating a snapshot. Essentially, a project in GitHub is made up of a bunch of commits. Basically a commit contain 3 pieces of information: information about how the files changed from previous snapshot, a reference to the commit that came before it (called the *parent commit*) and a hash code name to identify the commit.
- **Repo** refers the collection of all the files, the history of those files and all the commits. A repo can live in our local machine, referred to as a our **local repository**, or be located on GitHub's servers. In this case we talk about a **remote repository**.
- The action of downloading commits that don't exist on our local repository from a remote repository is called **pulling** changes. The process of adding our local changes to the remote repository is called **pushing** changes.
- In Git, a **branch** is a copy of all the files in your codebase. Think of the birth as a snapshot of the supporting branch, i.e. the branch from which . Each branch has an identifying name and its own set of version or commit history. When you create a new repository, the default branch is called `'main'` (previously called `'master'`). Even if you do not create any additional branches, you can performing all Git commands on that default branch.



(Illustration by Jordi Torres)

2. Getting started with Git

2.1. Installing Git

In order to intall Git you can visit the Git download page and run the installer for your operating system.

You can verify that Git is working from the command line:

```
$ git –version
git version 2.37.0
```

The installed version may be behind the latest one in your case, in particular if your installation is made from a Linux distro's official repository. Official repositories usually harbor stable but older versions.

We will be using the command line interface (CLI) from a terminal window to interact with Git repositories. There are also graphical user interface, or GUI, applications available for viewing and maintaining your repositories from Windows, Linux or MacOS. Those are not described here.

2.2. Initialize a local Git repository: `git init`

To start, let's create a new **local** directory called `localRepo` in `/srv/git/` and add a few files to it.

```
$ mkdir –p /srv/git/localRepo
$ cd /srv/git/localRepo
```

Create files `Readme.md` and `License.md`. Note that both are included in the root directory of many open source projects. These particular files use the markdown format. Markdown formatting on GitHub permits the production of ultra light-weight flat files for your prose (and perhaps also for your code) with a very simple syntax. See ***Appendix A*** for a quick overview on markdown.

```
localRepo $ echo '# README #' > Readme.md
localRepo $ echo '# LICENSE #' > License.md
```

Once inside `/srv/git/localRepo`, create a local repository with the following command. Commands will be followed by the output shown:

```
localRepo $ git init
Initialized empty Git repository in /srv/git/localRepo/
```

Internally a **`.git`** directory was added to local directory as you can check with:

```
localRepo $ ls –a
.        ..        .git       License.md      Readme.md
```

That turns that directory into a Git repository. This `.git` *hidden* directory will contain all of the configuration and metadata necessary for Git to keep track of our files and the changes that we make to them.

Store your **identity information**. It tells all Git repositories in our system our name/email, which will be applied to each commit. Type the following into the command line, replacing the fake identity with your own:

```
localRepo $ git config --global user.name "Cbhihe"
localRepo $ git config --global user.email insert-your-email-here
```

Additionally, you need to configure the **default push behavior**. There are two possibilities:

```
 localRepo $ git config --global push.default matching
```

or

```
 localRepo $ git config --global push.default simple
```

When `push.default` is set to '`matching`', Git will push local branches to the existing remote ones with the same name. Since version 2.0, Git defaults to the more conservative `'simple'` behavior, whereby it only pushes the current branch to the corresponding remote branch that `git pull' uses to update the current branch. If you are working on a centralized workflow (pushing to the same repository you pull from, typically `origin'), then you need to configure an upstream branch with the same name. This is the safest configuration for begineers.

Having initialized a directory as a Git repository, we can now start issuing other essential Git commands.

Warning: *Never (even try to) nest repositories. It will fail ... sometimes in unpredictable ways. You have been warned.*

2.3. Add & commit

Our local repository consists of three "trees" maintained by `git`:

- ***Working Directory*** which holds the actual files: *`localRepo`*. This is the root of your working tree.
- ***`index`*** which acts as a *staging area*, inside the working directory.
- ***`HEAD`*** which points to the last commit we have made, inside the working directory.

The first step in the basic Git workflow is to propose changes (add them to *`index`*). That is called *tracking* a file.

```
$ git add <filename>
```

or

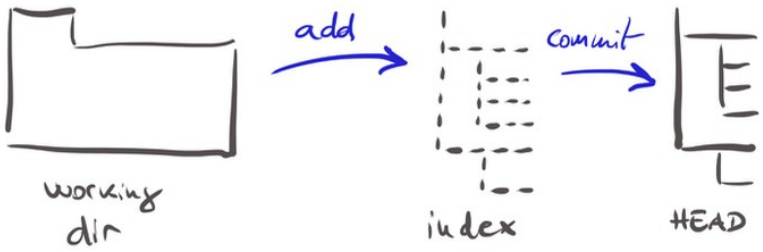
```
$ git add [-n] *
```

To perform a dry run, add the '*`-n`*' option. This may spare the unexperienced reader surprises.

To commit changes, do:

```
$ git commit -m "Type a short meaningful commit message here"
```

Now the file is committed and `HEAD` will reflect that fact, in the local working directory (but not in the remote repository yet).



(Illustration by Jordi Torres)

In our case we can execute:

```
localRepo $ git add Readme.md
localRepo $ git commit -m "My first commit... blah blah"
```

We do not add `License.md` file intentionally, in order to illustrate differences. The reason is sometimes Git repositories will have private data that you do not want shared (unintentionally added for instance). That is what for the `.gitignore` file is. This file is a list of files and/or directories that you do not want included in a repository. Git will not allow you to add any of the files referenced in the file `.gitignore` to a given repository.

You can also create a global **`.gitignore`** file, which is a list of rules for ignoring files in *every* Git repository on your computer. For example, you might want to create the file at `~/.gitignore_global` and add some rules to it. In that case pen a terminal window and run:

```
$ git config --global core.excludesfile ~/.gitignore_global
```

Some good rules to add to this file are included in ***Appendix B***. A detailed description of **`.gitignore`** is available in section 3.4 of this tutorial.

To list all the commits done so far between the last (local) *`main`* commit and the last (remote) *`origin/main`* commit:

```
localRepo $ git log [--oneline] [main  origin/main]
 …
 commit 1e5307f4d089ef9089775f671dd8cc48d29b2dcb
 Author: Cedric Bhihe <Cbhihe@users.noreply.github.com>
 Date:   Fri Apr 21 13:51:05 2017 +0200
 Update of Readme.md with emails
 ...
```

To see the **working tree status** (here on the *`main`* branch):

```
localRepo $ git status
 On branch main  Your branch is up-to-date with 'origin/main'.
 nothing to commit, working directory clean
```

Sometimes, you may need to **delete files**. After doing so locally, issue:

```
localRepo $ git add *
localRepo $ git commit -m "commit-message"
 On branch branch-name
 Changes not staged for commit:
 deleted: ...
 deleted: ...
...
```

This is expected behavior as otherwise any staged (local) changes including deletes would be committed automatically and subsequently pushed. To prevent spurious deletions of files on `origin` (i.e. on the the remote repo), Git actually requires a special command to stage the deletion of files, i.e. to add those deletions to the `index`.

```
localRepo $ git add -u
localRepo $ git commit -m "commit-message"
localRepo $ git push
```

From Git docs (git-scm.com/docs/git-add):

> Options for '`git add`': `-u` , `--update` *will updates the index just where it already has an entry matching <pathspec>. This removes as well as modifies index entries to match the working tree, but adds no new files. If no <pathspec> is given when -u option is used, all tracked files in the entire working tree are updated...*
> *Only matches against already tracked files in the index rather than the working tree. That means that it will never stage new files, but that it will stage modified new contents of tracked files and that it will remove files from the index if the corresponding files in the working tree have been removed.*

The above is strictly equivalent to the bash cmds:

```
localRepo $ git status | sed -n '/^# *deleted:/s///p' | xargs git rm
```

or

```
localRepo $ git ls-files --deleted | xargs git rm
```

2.4 Files' status and stage

- The states in which any file might find itself are:
  Untracked - when you first create the file, it is *untracked*. This happens specifically when newly created files

have never been staged before and do not appear in the local Git repo's previous snapshot (as part of the *index*). Files become tracked with both `'$ git add .'` or `'$ git add -A'`

- ***Staged / indexed*** - when you use `'git add'` command on the file, it goes in this area
  To visualize the name(s) of staged file(s): `'$ git diff --staged --name-only'`
- ***Committed*** - when you use the `git commit` on the file, it goes in this area
- ***Modified*** - the file is committed but has local changes, which are neither staged nor committed yet.

2.5 Removing already staged or committed files

Sometimes a file may accidentally be staged, committed to your local Git repo, or even mistakenly pushed to remote. How you deal with that depends on the context.

2.5.1 – Remove from staging area

To remove a file from the staging or cached area, issue:

```
localRepo $ git rm --cached <filename>
```

For whole directories and contents, use the `-r|--recursive` additional option:

```
localRepo $ git rm -r --cached <directory>
```

2.5.2 – Remove single file from committed area

The following applies to the latest *commit*, not to any *commit* already pushed to a remote repo. Remember that the most recent local commit is stored in ***HEAD***. Three commands are required in succession:

```
localRepo $ git reset --soft HEAD~1
```

to suppress the latest commit not already pushed to ***origin***, effectively reseting ***HEAD*** to its first parent, equivalently denoted: ***HEAD^*** or ***HEAD^1*** or ***HEAD~*** or ***HEAD~1***. At this point

```
localRepo $ git status
```

will reveal file references previously in `HEAD` s in the staging area (*index*). Remove them from the staging area with:

```
localRepo $ git rm [-r] --cached <filename>
```

or

```
localRepo $ git restore [-r] --staged <filename>
```

where the optional `` `-r' `` above is to apply the command to whole directories (and contents).

By running the above command, the file will appear in the untracked files' section. After having removed one or several file, go ahead and re-commit the remaining indexed files.

```
localRepo $ git commit -m "commit-message"
```

2.5.3 – Correct mistakes on commits and / or pushes

If a mistake is made after a commit but before a push, you may rectify the last commit, prior to pushing, with:

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

On the other hand if a commit mistake was made AND pushed, then you might need to ***force-push*** a second commit to clobber the first one:

```
localRepo $ git push origin/<remote-branch> <local-branch> --force
```

Warning:

- Force-pushing will overwrite any changes on the remote repo with the current local repo's state.
- The replaced commit will be effectively overwritten with a different SHA identification. This may have adverse consequences if collaborators in the remote repo's project have already either pulled the material erroneously pushed by you, or merged their own contributions to the project on the remote repo. In such cases, in order to be aligned with the new remote, others might want to reset their history locally.

There are 3 ways for other contributors, that is for all but the author of the force-push, to do that:
a) A ***hard reset*** will reset the index and working tree. Any changes to tracked files in other contributors' local working trees (staged files) since your forceful push will be lost.

```
localRepo $ git reset origin/main --hard
```

b) A ***soft reset*** will preserve others' staging areas, while updating their commit history with the content of your forceful push. It will also afford them the possibility to re-commit their changes to remote.

```
localRepo $ git reset origin/main --soft
```

c) However it is generally preferable to proceed with a much safer interactive ***rebase***. "Rebasing" consists for other contributors in replaying their local commits on top of any other pushed commits, with:

```
localRepo $ git rebase -i origin/main
```

This will invoke *rebase* in interactive mode where users can choose how to apply each individual commit not in the remote's commit history on top of which they are rebasing.

If the remote's commits removed (with `$ git push -f') have already been pulled into the local history of any contributor about to conduct a rebase, then those commits would be listed as valid rebase commits, and would be reapplied. To prevent this, i.e. in order for those commits not to simply be re-included into the history for the branch, they would need to be deleted as part of the rebase, lest one wishes to see them reappear in the remote history on the next push.

Users can limit their interactive rebase scope, with:

```
localRepo $ git rebase -i HEAD~X
```

where X is the number of commit log entries before the force push, i.e. the number of previous commit generations before force push. Git will "collect" all the commits in the last X log entries and if there was a merge somewhere in between, that range will see all those merges' commits as well, so the outcome will really be X+ commits to manually approve / edit.

To better understand what your options are in complicated situations where the project time-line includes one or more ***merges***, between the correction and the original commit and push, consult the help with `$ git <cmd> --help', see related posts on StackOverflow, or visit:

git-commit(1) man page
git-rebase(1) man page
git-push(1) man page

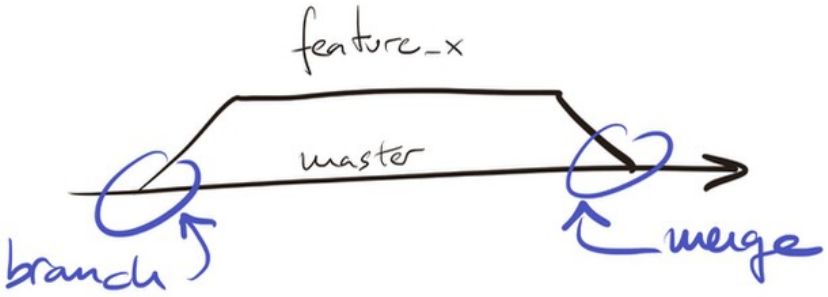
also available from your Linux console as `$ man git-commit', etc.

2.6. Branching

Branching is a powerful mechanism in any repository tool that allows developers to veer off into a tangential or experimental direction, without affecting the main codebase.

To create a branch, we **fork** it from another existing branch, likely from `'main'`. In this tutorial we choose to call the branch from which a new branch is forked, the *supporting branch*. Then we switch to the newly forked branch and start working inside it. The new ”feature-x” branch will contain all the historical changes accumulated in the supporting branch up to the point of the fork. It means that the two branches do not stay in sync automatically when applying subsequent changes as one goes ahead with development work or any other modification it may undergo.

In case a forked branch must be discarded, simply delete it. On the other hand, if we want to keep the code we can **merge** commits from any branch into our `main` branch.



(Illustration by Jordi Torres)

To **create a new branch locally**, name it "feature_x" and switch to it, issue:

```
$ git checkout -b feature_x
```

To **switch back to main locally**, do:

```
$ git checkout main
```

To **visualize difference locally** between the new branch and its source (here the “main” branch), do:

```
$ git diff main feature_x
```

To **visualize differences remotely** between a branch and its upstream-source (here the “`origin/main`” branch), do:

```
$ git diff origin/main origin/branch
```
or
```
$ git diff ...origin/branch
```

where the default name “origin” points to the repo on GitHub. (We will see that that default can be changed.)

To **delete a branch locally**, do, from another branch, possibly from `main`:

```
$ git branch -d <branch>
```

Replace `'-d'` with `'-D'` above to force deletion.

One can also delete a branch on GitHub/Lab via the GUI, or from your local console, with:

```
$ git push origin :<branch>
```

where `':'` above indicates deletion of the named branch.

2.7. Remote branches: fetch, merge and pull

From any local repo, you can:

- **list** all the remote connections that current repo has to other repositories:

```
localRepo $ git remote [-v]
```

The `-v` flag (verbose) includes the URL of each connection. For instance:

```
origin  https://github.com/Cbhihe/CLC_cloud-computing.git (fetch)
origin  https://github.com/Cbhihe/CLC_cloud-computing.git (push)
```

- **create** a new connection to a remote repository specified by its <repo_url>:

```
localRepo $ git remote add <new_name> <repo_url>
```

After adding a remote, you'll be able to use <new_name> as a convenient shortcut for <repo_url> in subsequent Git commands. In this case however, the term "origin", previously meant to denote the remote repo, is no longer functional. It is replaced by the new name <new_name>. For instance:

```
localRepo $ git remote add <new_name> <repo_url>
<new_name>      <repo_url> (fetch)
<new_name>      <repo_url> (push)
```

Note that you can also acccess and manually edit that info with your terminal session's default editor, by issuing:

```
localRepo $ git config -e
```

The config file is at: `localRepo/.git/config`

- **remove** the connection to the remote repository called <new_name>:

```
localRepo $ git remote rm <new_name>
```

This will not remove the remote repo, just its local shortcut, <new_name>, previously created to alias its url <repo_url>.

- **rename** a remote connection from <old-name> to <new-name>:

```
localRepo $ git remote rename <old_name> <new_name>
```

- **fetch** will import commits (download objects and refs) from the remote repo in the local repo:

```
localRepo $ git fetch <remote_repo> [<remote_repo_branch>]
```

Example 1:

```
localRepo $ git fetch origin
```

will fetch all branches at origin (remote repo) and will display their commits up to the time the fetch cmd was issued.

Example 2:

```
localRepo $ git checkout 2nd-branch
localRepo $ git fetch https://github.com/Cbhihe/syncThat.git 2nd-branch
```
or
```
localRepo $ git fetch origin 2nd-branch
```

The resulting commits are locally stored as *remote branches* instead of the normal local branches. This gives the user a chance to review changes before integrating them into his/her copy of the project.
Note that specifying a <branch> has the effect of only downloading info specific to said branch of the remote repo.

Remote branches are just like local branches, except they represent commits from somebody else's repository. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (similar to checking an old commit). You can think of them as read-only branches.

Caution: You cannot fetch a remote branch if locally you already have an identically named branch.

To **view existing branches**, simply issue the cmds:

```
localRepo $ git branch -r       (← for remote branches)
  origin/feature_x              (← example)
  origin/2nd-branch             (← example)
  origin/main                   (← always present)

localRepo $ git branch          (← for local branches)
  feature_x
  2nd-branch
  main
  remote

localRepo $ git branch -a       (← for all branches)
  feature_x
  2nd-branch
  main
  remote
  remotes/origin/feature_x
  remotes/origin/2nd-branch
  remotes/origin/main
```

Remote branches can be inspected with the usual *git checkout* and *git log* commands:

```
        $ git log -r
or      $ git log origin/main
```

Further, all local repos can be updated with one command by issuing:

```
$ git remote update
```

or equivalently

```
$ git fetch -all
```

Both will import commits (download objects and refs) from all remotes, not just one as with `*git fetch origin*' from within a specific local repo...

Once a branch's commits are fetched and approved, you can merge it with the active branch, that is with the branch on which you are currently checked out (main in the example below):

```
localRepo $ git checkout main
localRepo/main $ git merge <branch>
```

where `localRepo/branch` can be the local main or any branch to which <other_`local_branch`> is being merged. The above command must be issued from ***outside the branch*** (`<local_branchname>`) to be merged.

Git tries to auto-merge changes. Is is not always possible and may result in *conflicts*. You are responsible to resolve those conflicts manually by editing the files shown by `git`. After manual changes are made to resolve conflicts in a file, mark the file <`filename`> as staged with:

```
localRepo/branch $ git add <filename>
```

Again, before merging changes, you can also preview them by using:

```
localRepo $ git diff <source_branch> <target_branch>
```

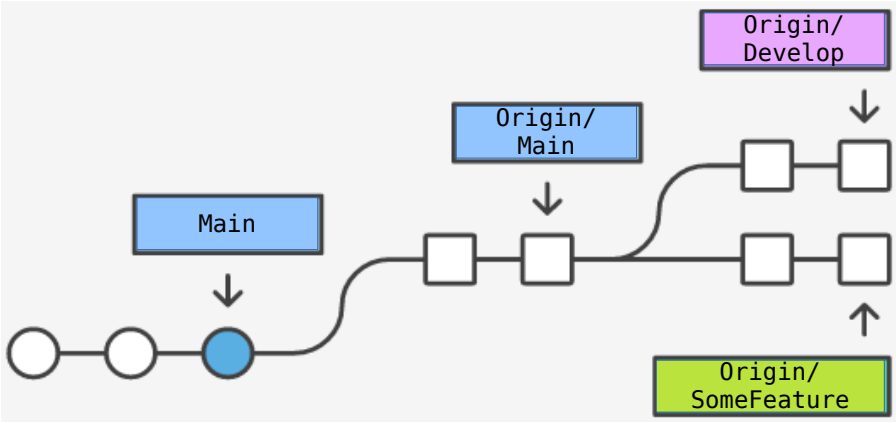
So, unlike with SVN[i], synchronizing your local repository with a remote repository is actually a two-step process: fetch → merge. The *`git pull`* command is a convenient shortcut for this process.

Imagine the situation illustrated by the figure below. To see what commits have been added to the upstream main, run *`git log`* using `origin/main` as a filter:

```
$ git log –oneline   main origin/main
```

To approve the changes and merge them into your local `main` branch with the following commands:

```
$ git checkout main
$ git log origin/main
```



(Source: unknown)

followed by:

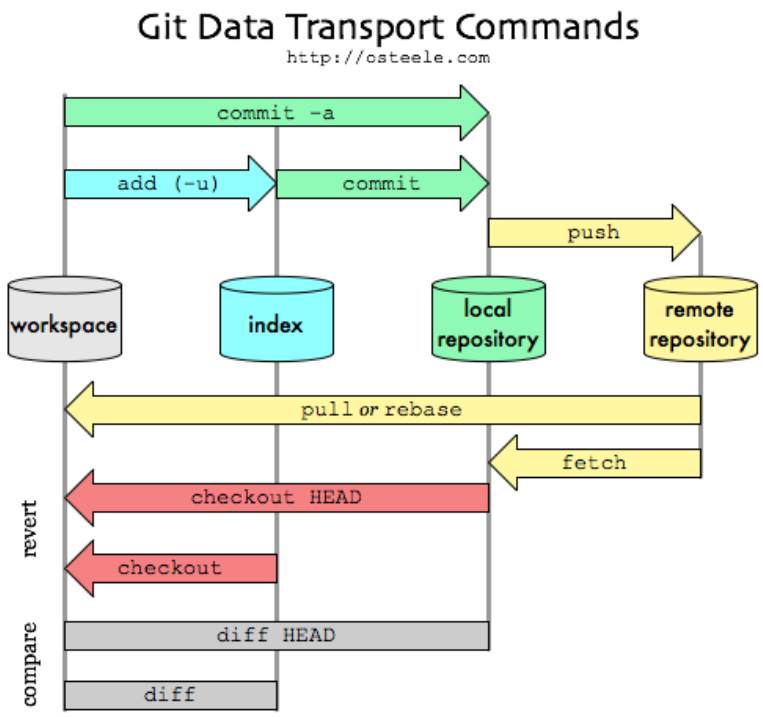
```
$ git merge origin/main
```

The `origin/main` and `main` branches now point to the same commit, and you are synchronized with the upstream developments.

What if youwanted to merge a remote branch called `test` with `origin/main`:

```
localRepo $ git checkout main                  (← go to your local main)
localRepo (main) $ git pull origin main        (← update your local main )
localRepo (main) $ git merge test              (← merge local branch test with local main )
localRepo (main) $ git push origin main        (← pushed local main to remote main )
```

i SVN: short for subversion, which is a type of version control system. More at http://aymanh.com/subversion-a-quick-tutorial



(Source: http://blog.osteele.com/2008/05/my-git-workflow/)

2.8. Renaming branches

2.8.1 – On the remote repo (at "origin")

On https://www.github.com:

- navigate to the main page of the repository.
- above the list of files, click **Branches** and select the branch to be renamed from the displayed list.
- to the right of the branch you want to rename (e.g. "main"), click on the small pencil-shaped icon to edit.
- type in the new branch name (e.g. "main") and click the *"Rename"* button.

2.8.2 – On the locally cloned repo

Update it with the changed branch name(s) by running the following commands:

```
localRepo $ git branch -m master main          # short option `-m' stands for "move"
```

Then sync your locally renamed branch with the previously renamed remote branch.

```
localRepo $ git fetch origin
localRepo $ git branch -u origin/main main
localRepo $ git remote set-head origin -a
```

2.9. Renaming repos

Renaming a Git repo may mean renaming the local repo's directory name, or the remote repo's name. Each requires different steps.

2.9.1 – Renaming the remote repo ("origin")

- Go to your distant repo (named '`origin`' by default), and locate the [Settings] tab or section. Change its name and note the changed repo URL.

- Open a local host terminal, and navigate to your local repository directory to set the changed remote repo's URL:

```
localRepo $ git remote set-url origin git@github.com:<user>/<new_name.git>
```

or, if you use a public key to access your remote repo:

```
localRepo $ git remote set-url origin git@HOSTNAME:<user>/<new_name.git>
```

where `HOSTNAME` is referenced in your local `~/.ssh/config`.

2.9.2 – Renaming the local repo

- rename the corresponding directory, using `mv` from the CLI.
- in case submodules are present, an additional corrective step is mandatory. As Git uses an absolute path to index submodules (when they exist), only renaming the local repo directory will impede further access to submodules. Each submodule configuration file, located at `.git/modules/config`, has the form:

```
[submodule "old/path/to/submodule"]
   path = old/path/to/submodule
   url = https://github.com/user/projectName.git
```

To fix that issue, it is necessary to edit each submodule's path information manually with:

```
localRepo $ find . -type f \( -name ".git" -o \( -path "*.git/modules/*" -name
config \) \) -print0 | xargs -0 grep --colour "old/path"
localRepo $ find . -type f \( -name ".git" -o \( -path "*.git/modules/*" -name
config \) \) -print0 | xargs -0 sed -i -e "s#old/path#new/path#g"
```

2.10. Removing / deleting branches

Git branches reflect the state of a project with a commit history line that diverges from the branch from which it forked. Typically when a branch is merged with the branch from which it was forked, commit histories are reconciled and the merged branches stay behind, while the codebase moves on with later developments and modifications. Ultimately merged branches may clutter the project. For that reason it may be useful to delete certain branches after merging them.

To list local, remote and all branches, the commands are respectively:

```
localRepo $ git branch              # local
localRepo $ git branch -r           # remote
localRepo $ git branch -a           # all (local + remote)
```

2.10.1 – Deleting a local branch

This can only be done from outside the branch which is to be deleted. After that branch's commits are fully merged, do:

```
localRepo $ git branch -d  local_branch_name
```

If you want to disard that branch's commits, and force-delete it, simply do:

```
localRepo $ git branch -D  local_branch_name
```

2.10.2 – Deleting a remote branch

From your local cloned repo, do, in case your remote repo's name defaults to "`origin`":

```
localRepo $ git push origin -d remote_branch_name
```

2.11. Fetching remote branches

New Git branches may appear on the remote repo, "`origin`". That may be explained by conributors pushing new branches to the remote repo in order to push commits to it. You in turn may be interested in fetching such a remote branch, say "`new_rem_branch`", on your local repo. One of the ways to do this is:

```
localRepo $ git fetch
localRepo $ git checkout -b new_rem_branch --track origin/new_rem_branch
```

In the above "`-b new_rem_branch`" can be omitted, in which case the new local branch name will default to the new remote branch name.

3. Getting started with GitHub

3.1. Creating a GitHub account

Github offers free accounts for users and organizations working on public and open source projects, as well as paid accounts that offer unlimited private repositories.

- Set up your GitHub account. Within the next half hour, you will know how to use it.

Whenever you are logged in to the GitHub website, you will see a plus icon in the upper right corner. When you click on that, you will see a menu that includes a *New Repository* link. Click on that link. Choosing a repository name comes next (if we have spaces in our repo name, GitHub will automatically replace them with hyphens).

In this hands-on, we are going to create a remote repository on GitHub where we can push the local repository that we created previously (our local and remote repos do not have to have the same name). We can decide if our repo is public or private.

The next important thing we need to look at is the *Initialize this repository with a README* checkbox. This is a very important step in creating our remote repo. If we check the box, it will automatically create and commit a *README file* in our remote repo. We will usually create our local repo first, so we will not check this box in this case (this is the case of our example in this hands-on).

What you get is the screen shot below.

3.2. Connecting a local repo to a Github repository

Now you should see instructions (similar to the following screenshot) on how to connect your local repository in your computer to the remote one you just created on GitHub.

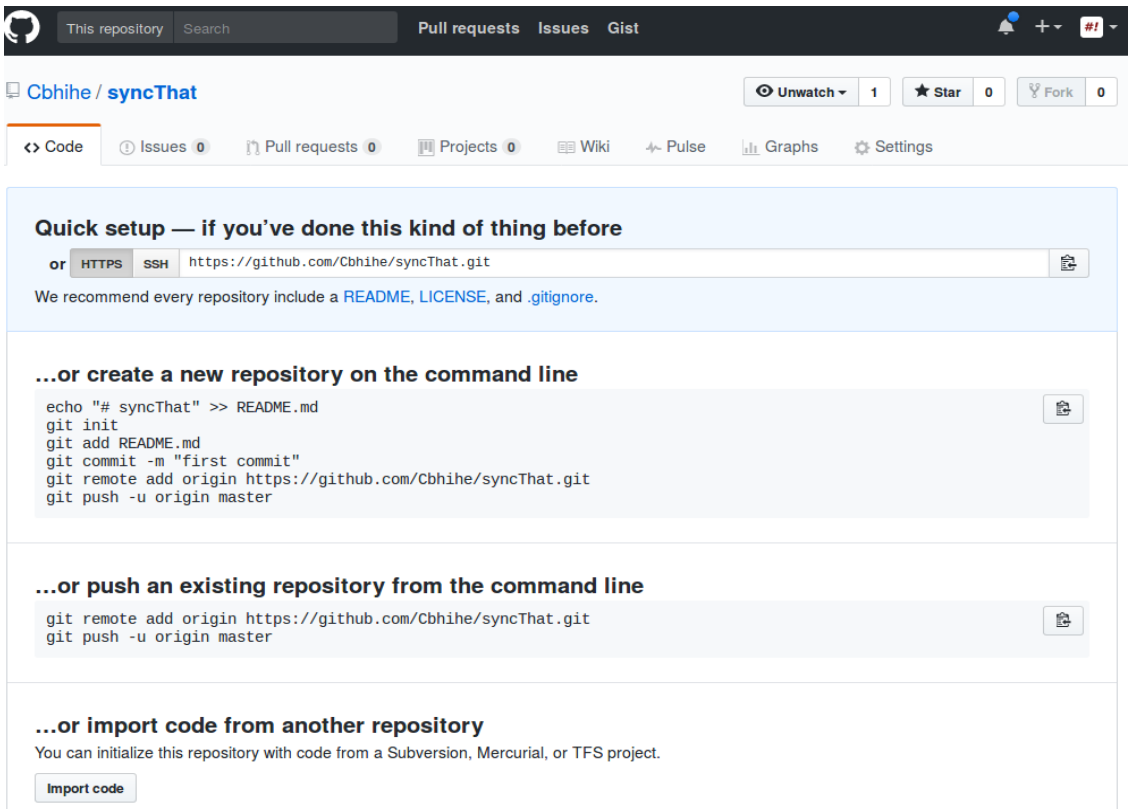
Looking at just the first option, the set of Git commands should look very familiar, we have already done the first four commands. We only need to perform the last two commands in order to link our local repository with the remote one.

Remember that your changes are in the `HEAD` of your local working copy. To send those changes to your remote repository, your local repo must know about the remote repo on GitHub.

For that add a reference to that remote repo with `` `git remote add origin <server>' `` from your local repo. Only now can changes be pushed to the selected remote server:

```
localRepo $ git remote add origin https://github.com/Cbhihe/clc.git
```

In the above *`origin`* is an alias or place-holder for the remote repo.



Before we move on, let's take a look at the command we just executed. The primary Git command we issued is `git remote' to work with remote repositories.

```
localRepo $ git remote -v
```

By itself the above would list all remote repositories known to the local repo. If we pass additional sub-commands to the `git remote' command, we can further add, remove, and modify the remote GitHub repository that your local repository is linked to.

Parameters:

```
origin
https://github.com/Cbhihe/clc.git
```

refer to the remote repository. In particular `origin` is an alias that we will use locally to interact with this remote repository and `https://github.com/Cbhihe/localRepo.git` indicates the remote repository's URL.
You can open `.git/config` with a text editor to see how Git stores the information you just added.

You are now ready to **push** our code to GitHub with:

```
$ git push -u origin main
```

`origin` is an alias or place-holder for the remote repo and *`main`* indicates the remote repository branch. (It could be any branch beside *`main`*) The *`-u`* option shows untracked files (it includes in particular files that were removed from the local repo from the console shell.

If we have never pushed to GitHub before, we will now be seeing a login prompt, and we are required to introduce our GitHub username and password. SSH secure login can also be arranged as is detailed herafter in Section 3.5. But if you follow this tutorial and come from the previous sections, your output should as follows:

```
Username for 'https://github.com': Cbhihe
Password for 'https://Cbhihe@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 236 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/Cbhihe/clcLab.git
 * [new branch]      main -> main
Branch main set up to track remote branch main from origin
```

After a successfully login, a message appears to the effect that a number of objects have been written. We will also see a message saying that a local main branch was set up to track the remote main branch (thanks to the `-u` flag). As a result the remote GitHub account now holds files previously added with `` `git add' `` in the local repo: `Readme.md`

Sometimes the code in our remote repository will contain commits we do not have in your local repository (for example because we work on a team or we made a change to a file directly on GitHub.com). In those situations, we will need to **pull** the commits from the remote repository into our local repository with the command:

```
localRepo $ git pull origin main
```

3.3. Branching

It is important to be aware that branches can exist both locally and on remote repositories. Of course a local branch, being local, is not available to others unless it is pushed to the remote repository with:

```
$ git push <remote> [<branch-name>]
```

You can omit to specify the branch name, if you are in that branch as you issue the command. However you must specify the branch name if you specifically need not to push other branches to the origin.

If pushing everything on the local repo to the remote repo (e.g. ***`origin`***) is ok, then do:

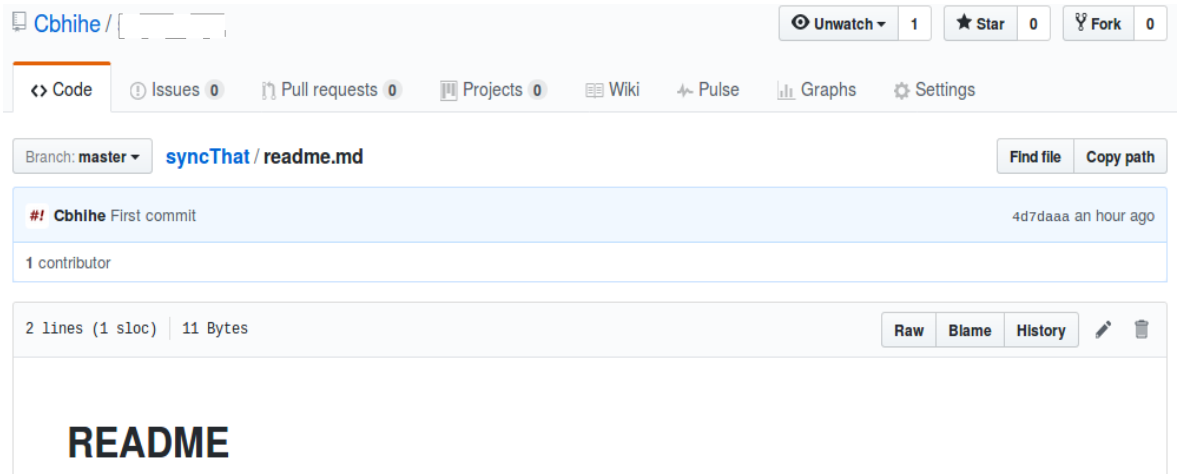
```
$ git checkout main ; git push <remote> --all
```

To update the local repo with the newest commit(s) on the remote repo, go to your local repo's working directory. *Fetch* and *merge* remote changes with:

```
$ git pull
```

To merge another branch into your active branch (e.g. ***`main`***), use:

```
$ git checkout main
$ git merge <branch>
```



3.4. Exclude files when pushing to a remote repo with "`./.gitignore`"

Because your local repo is the place where you carry out your developments, a lot of different files may be generated as an intermediate by-product of that work-in-progress. Think about all those files that a typical preprocessing, compilation and linking may produce as part of a typical build pipeline. A simple but good illustration of that would be object files in the C or C++ language(s), that result from the compilation of several source files. That step is an intermediate one, since those object files are subsequently linked to produce one of more useful executable(s). Many times not all or none of the object files produced during compilation are needed by the final recipient(s) of the executable(s). For that reason you may want to ignore those `*.o` files from the sync process between your local repo, where they live happily, and the remote repo, where you decided that they have no business being at all.

In the end the above description shows it may not always be necessary to commit and sync everything between local and remote repo. When looking at the remote repo (the "origin" to which you push), others will not have to deal with clutter, or with files you want to keep unseen. Those might include files with secret-keys or any other type of file you do not want to share with the project collaborators or with the wide world.

How can we ensure that in a somewhat automated fail-safe way with `git`?

Enters the file `.gitignore`. It contains directives to include or ignore files in the syncing process between local and remote git repos. More precisely a `.gitignore` file specifies intentionally *untracked files* that Git should ignore. An *untracked file* is one that was not staged and subsequently committed.

File exclusions and inclusions can be specified with simple patterns according to a specific grammar and syntax described in sub-section 3.4.3.

3.4.1 – Where to best place commit exclusion patterns on a host endowed with Git

There maybe as much as three instances of commit-exclusion directives, each with a differentiated objective:

(a) In a local repo an exclusion pattern directive specified in a `.gitignore` file is version-controlled like any other file not excluded from being trackingwill be propagated to the remote repo and to other contributor's local repo.

Such a `.gitignore` file generally sits in the repo's root directory. If your local repo consists of a directory tree with multiple levels, you can have a unique `.gitignore` file sitting at the local repo's file tree root and all levels below will inherit its directives. This holds unless you also include some additional `.gitignore` file somewhere down the repo's file tree. If you do, then a file positioned in any tree branch lower than the tree root will inherit higher directives and supersede those higher ones it contradicts (the ones closer to the repo's root directory). The resulting directives will in turn be passed on to the tree branches below it (if any).

Any directory in your repo's working tree may contain only one `./.gitignore` file. Note that in a local repo `./.git/` is not part of the working tree, i.e. the actual file tree where your version-controlled directories, files and their contents live with their latest modifications. In other words `./.git` is effectively excluded from version-control.

(b) In a local repo certain exclusion pattern directives may also concern only that particular local repo. For instance they can be auxiliary files that live inside the repository but are specific to one user's workflow for that specific repo. In this case such directives should go into the `./.git/info/exclude` file of the local repo. As mentionned before that file is not subject to version-control and is therefore propagated to neither the remote repo nor to other contributors working with that remote repo.

`(c)` You may also specify a global `.gitignore` file that provides a default set of directives system-wide and for all local repos in a given host. As for (b) those directives are not propagated. Its name and location do not matter as you need to validate it for it to become your global exclude-file. If you choose to place it in your home directory and call it `.gitignore_global`, then you will need to issue:

```
$ git config --global core.excludesfile ~/.gitignore_global
```

This above command directly modifies the contents of the section "[core]" of file ~/.gitconfig which becomes:

```
excludesfile = $HOME/.gitignore_global
```

You can also avoid creating that new global .gitignore file, by resorting to Git's global ignore default located on any *nix host at ~/.config/git/ignore. If you so choose, just ensure that, as in the above, the section "[core]" of file ~/.gitconfig coincides with that location choose. In the latter case it should read:

```
excludesfile = $HOME/.config/git/ignore
```

which you can specify by hand with your favorite text editor, or equivalently by issuing the previous command:

```
$ git config --global core.excludesfile ~/.config/git/ignore
```

Just creating a .gitignore file (global or not) in the right place is not enough. You also need to populate it with exclusion patterns per your needs. Reading the next section will help you in that task.

3.4.2 – How to subject an already staged or committed file to exclusion directives

Remember that files already tracked, i.e. already staged, by Git are not affected by any .gitignore directive. So how can one subject those already tracked/staged files to a .gitignore directive ? Simple. Just stop tracking files that was previously staged, i.e. placed in the tracking index with the command `git add <filename>'. For that, visualize tracked files with:

```
$ git status       # output: filename1 filename2 directory1
```

then remove them from the index:

```
$ git rm --cached <filename1 filename2 ...>
```

Add the option -r to the above command, to also remove directories and their contents:

```
$ git rm -r --cached <filename1 filename2 directory1 ...>
```

If the tracked files were also committed, and they are the last commit, they show in the HEAD. In that case, untracking them consists in first removing them from the HEAD, then, as shown before, from the index.

```
$ git reset --soft HEAD~1
$ git rm --cached <filename>
```

Note that sometimes you will see the tilde (~) in the first command above replaced with a caret (^). Both coincide with the first parent of the commit currently in HEAD.

To understand the significance of the long option `--soft', consult the man page with `$ man git-reset'. In essence the `--soft' option ensures that the reset command touches neither the index (i.e. the staging area) nor the working tree. So the first command takes the latest commit away but resets neither the staging area not the working tree. Hence the need for the second command to remove tracked files from the index (i.e. the staging area).

There is another way to proceed, using `git restore …'. Consult the manual page with `$ man git-restore'.

3..4.3 – Creating .gitignore exclusion patterns

Dom Habersack has good advice on that front:

1/ Toptal's .gitignore tool will helps you create a .gitignore file based on your choice of OS, code editor, and more.

2/ If you prefer to roll your own, you may look for inspiration in GitHub's very own gitignore repository.

My advice, however is to start with consulting your host's manual page, before experimenting.

```
$ man 5 gitignore
```

Here's a simple example with globbing patterns to get you started along that path. Imagine a local repo with the following file tree:

```
myRepo
   |-- 01_parse_and_sort.ipynb                T
   |-- 27_<...>.ipynb                         T
   |-- <...>                                  NT
   |-- Data/                                  T
   |     |-- cand_text                        NT
   |     |-- ref_text                         NT
   |     |-- machine_translation              T
   |     |-- human_translation                T
   |     |-- <...>                            NT
   |
   |--.git/                                   NT
   |-- .gitignore                             T
   |-- .ipynb_checkpoints/                    NT
   |-- License.md                             T
   |-- .python-version                        T
   |-- nlp_module.py                          T
   |-- Readme.md                        T
   |-- secret-keys                            NT
```

What we need is to ensure a parsimonious version-control applied to only certain files (earmarked on the right with "T"), excluding all the others (earmarked on the right with "NT") from tracking. Note for instance that we want to include the subdirectory `./Data/` , excluding all files within but `./Data/machine_translation` and `./Data/human_translation`.

To achieve that our `./.gitignore` file located in the root directoy could be written in several ways. One is:

```
$ cd /path/to/myRepo
$ cat ./.gitignore
*
!/[0-9][0-9]_*.ipynb
!/Data/
!/Data/*_translation
!/.gitignore
!/*.md
!/.python-version
!/*.py
```

All patterns (one per line) have paths relative to the location of the `.gitignore` file within the working tree.

- 1st line: `*`, a globbing pattern that excludes everything, files and directories alike on the same tree level as that of the `.gitignore` file. Later lines that start with the optional prefix '!' reverse that exclusion (i.e. negate the previous exclusion pattern) for the path pattern that follow the exclamation mark. In this example all subsequent lines do just that.

- 2nd line: negate the previous blanket exclusion for all `ipynb` files whose names start with 2 digit and an underscore,

- 3rd line: include the directory `./Data/` but nothing inside,

- 4th line: include all files whose names end with "`_translation`" in the `./Data/` directory,

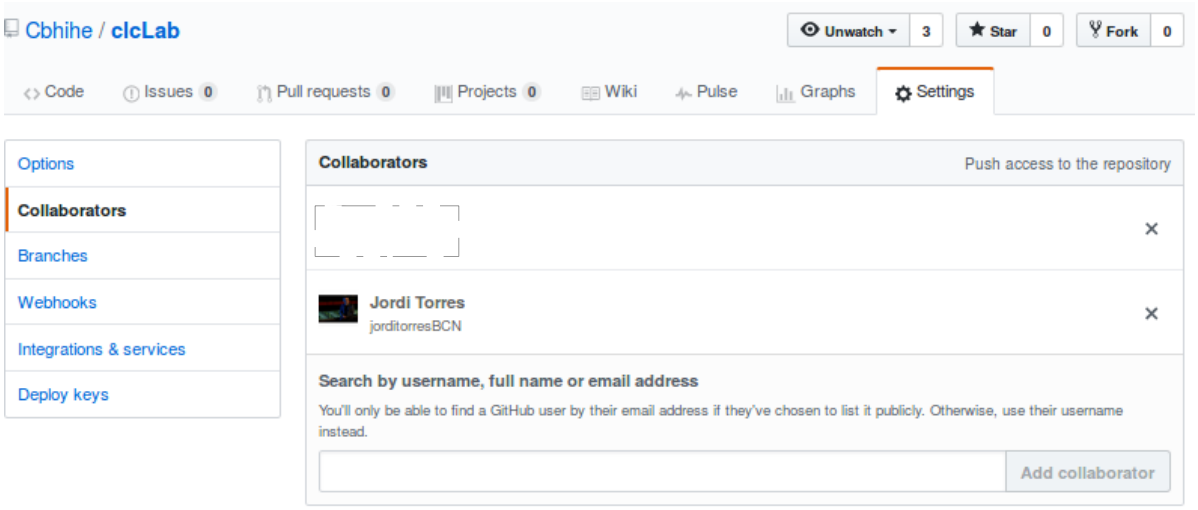
- 5th to 8th lines: include listed filenames including all those whose names end in `.py` and `.md`.

3.5 Inviting collaborators to a project

Hopefully you will not work on your own on all your project. Chances are that you will eventually want to invite outsiders (also with a GitHub account) to share your project. In that case “sharing” means that the invitee will acquire the capacity to act on your project’s content and flow, much in the way that you would.

In order to do so, log onto GitHub and go to the concerned project’s repo.

- go to Settings inside your project’s repo
(Settings are located on the right hand side of the horizontal menu above the repos contents)
- on the left (vertical menu) clik on “Collaborators”
- add colaborators to your project, by filling in their full names or their GitHub’s nicknames.
- an invitation will be automatically sent to them.



In the case depicted in the previous figure, two collaborators have accepted GitHub-user Cbhihe’s invitation and are featured.

3.6. Securing an SSH connection to your GitHub account

You can use either SSH private and public keys of GPG keys to connect to a GitHub account without entering a username and password everytime you push something from the local repo. What follows is a step by step guide to acheive that. (Also see https://help.github.com/articles/connecting-to-github-with-ssh/)

Open Terminal:

```
$ ls -al ~/.ssh        # Lists all files in the .ssh directory
id_rsa
id_rsa.pub
...
```

Check the directory listing to see if as shown above, you already have a public SSH key. By default, public keys filenames are id_rsa and id_rsa.pub.

3.6.1 – Creating an SSH key

If you already have a public key you would like to use, skip to Section 3.5.2 to directly add that key to the ssh-agent.

If you don't have an existing public and private key pair, or don't wish to use any that are available to connect to GitHub, then generate a new SSH key.

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

The above creates a new ssh 4096 bit key, using the provided email as a label. You are then prompted to "Enter a file in which to save the key". Pressing 'Enter' accepts the default file location. At the following prompt, type a secure passphrase. The passphrase can be empty, although it is not a secure practice. The newly created key consists in reality of two files. One contains the public key and ends with the suffix "`.pub`" and is called "`id_rsa.pub`" by default. The offer file, called "`id_rsa`" by default, contains the SSH private key.

3.6.2 – Adding an SSH key to the SSH-agent

If you see an existing public and private key pair listed (for example *id_rsa.pub* and *id_rsa*) that you would like to use to connect to GitHub, you can add that SSH key to the ssh-agent.

Start the ssh-agent in the background.

```
$ eval "$(ssh-agent -s)"
Agent pid 59566
```

Add the SSH private key to the ssh-agent. If you created a key with a different name, or if you are adding an existing key that has a different name, replace *id_rsa* in the command with the name of the desired private key file.

```
$ ssh-add ~/.ssh/id_rsa
```

Optionally you can check that the key was correctly added to the SSH-agent:

```
$ ssh-add -L            # for full key
$ ssh-add -l            # for fingerprint
...
```

or

The result of the command above will show you either the full key (with '`-L`'), or a fingerprint (with '`-l`') you should check for correctness. If everything checks, you are now ready to add the SSH key to the GitHub account.

3.6.3 – Adding an SSH key to a GitHub account

To configure your GitHub account to use your new (or existing) SSH key, you must first add it to your GitHub account.

- copy the SSH public key to your clipboard:
- locate the `~/.ssh` folder
- open the public key file whose default name is "`id_rsa.pub`" (if you have not called it something else),
- paste its contents, including the starting tag ("ssh-rsa") at the beginning, but not the end tag, separated from the key by a single space. Do not add any newlines or whitespace. Do not modify the contents of that file in any way.
- in the upper-right corner of any GitHub page in the account of interest, click the profile photo, then:
    - click ***[Settings]***
    - select ***[GPG and SSH keys]***
    - click ***[New SSH key]*** or ***[Add SSH key]***
    - add a descriptive label for the new key in the "Title" field
        e.g., if your host is called ***Mozart***, you might call this key "Mozart SSH key".
    - paste your key into the "Key" field
    - click ***[Add SSH key]***
    - if prompted, confirm your GitHub password.

3.6.4 – Test the SSH connection

To test the SSH connection, do in terminal:

```
$ ssh -T [-v[v[v]]] git@github.com
.... (message from GitHub)
```

Similarly for GitLab, you would do:

```
$ ssh -T [-v[v[v]]] git@gitlab.com
.... (message from GitLab)
```

Verify that the fingerprint in the message you see matches one of the messages in step 2, then type "`yes`".
Verify that the resulting message contains your username.

3.6.5 – Troubleshooting

After generating a SSH key-pair and adding the public-key to GitHub, pushing any change may result in GitHub still demanding the username/password combination. Git Hub does so probably because the `origin` (on GitHub) point to the https url rather than to the ssh url. To reveal that, do in terminal:

```
$ git remote -v
$ git remote set-url origin git@github.com:<Username>/<Project>.git
```

The first time a push is made, GitHub will normally request the corresponding passphrase.

4. Git clone

So far, we have been talking about connecting an existing remote repository with an existing local repository. But what if we don't have an existing local repository, and just want to pull down all the contents of a remote repository? In this case we can issue `` `git clone` `` to clone the contents of an existing remote repo in our local repo. That cmd has several options, but most of the time we will only consider its basic usage `` `git clone` ``.

`` `git clone` `` is used mainly when we need to work with a remote repo, and do not yet have a local repo created. This is the case of many of the hands-on suggested in this course. For that reason `` `git clone` `` is used in almost every lab's hands-on session. For instance, assume that the documentation of today hands-on is at `https://github.com/Cbhihe/localRepo.git` . Here's what it looks like in action:

```
~ $ pwd
/home/Cbhihe
~ $ git clone https://github.com/Cbhihe/clcLab.git hands-on_lab
Cloning into 'hands-on_lab'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
~ $ ls
hands-on_lab
~ $ cd hands-on_lab/
hands-on_lab $ ls -a
.        ..        .git      README.md
hands-on_lab $
```

As you can see a new local git repository was created in a directory called `hands-on_lab`. The full content of the remote repository `https://github.com/Cbhihe/localRepo.git` was downloaded into the newly created local repo.

Finally, indicate that a remote repo was added to the local repo's config, pointing to the remote repo URL, aliased as `origin`. This is the same thing as issuing:

```
$ git remote add origin https://github.com/Cbhihe/clcLab.git
```

from within the local repo's directory.

5. To go into more detail

A good reference is *Pro Git*, the book written by Scott Chacon and Ben Straub and published by Apress. It is available here (pdf, epub, mobi, html). Its contents are licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. A dead-tree printed version of the book is also available (for a price) on amazon.com.

Appendix A

Markdown format

- To create a heading, add one to six # symbols before your heading text.

  The size of the heading is set in inverse proportion to the number of # used.

- To indicate emphasis with bold use ** before and after the selected text.
- To indicate emphasis with italic use * before and after the selected text.
- You can quote text with a >.
- You can call out code or a command within a sentence with single backticks `. The text within the backticks will not be formatted.
- To format code or text into its own distinct block, use triple backticks.
- We can create an inline link by wrapping link text in brackets **[ ]**, and then wrapping the URL in parentheses **( )**.
- We can make a list by preceding one or more lines of text with **-** or * . To order the list, precede each line with a number.
- We can create a new paragraph by leaving a blank line between lines of text.

For more information, see Daring Fireball's "Markdown Syntax".

Appendix B

Example of `.gitignore` file

```
# Compiled source #
###################
*.com
*.class
*.dll
*.exe
*.o
*.so

# Packages #
############
# unpack files and commit raw sources as Git has its own compression methods for storage
*.7z
*.dmg
*.gz
*.iso
*.jar
*.rar
*.tar
*.zip

# Logs and databases #
######################
*.log
*.sql
*.sqlite

# OS generated files #
######################
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trash
ehthumbs.db
Thumbs.db
```