GitHub: a Practical Introduction

François Gerard

October 13, 2018

Abstract

This is a very short and practical introduction to GitHub intended to future RAs and co-authors. Most of commands for GitHub are very well documented online and you should have no problems to find them, but what can be sometimes tricky is understanding how to start using GitHub. I will briefly describe the main concepts of GitHub and try to illustrate the most important tools using one practical example. After reading this introduction you should be able to use GitHub in a project straight away. The idea is to give you an understanding of the main tools, so that you can later get a deeper understanding of them through online documentation. I recommend several website for specific tools along the way, but the most complete reading is Pro Git - Everything You Need to Know About Git

1 Introduction

In the following sections I will guide you through (i) editing an existing project in Git and (ii) creating a project of your own and transferring it to Git. Although the examples are very simple, the commands are the same that you'll be using in more complex projects.

2 GitHub Basics

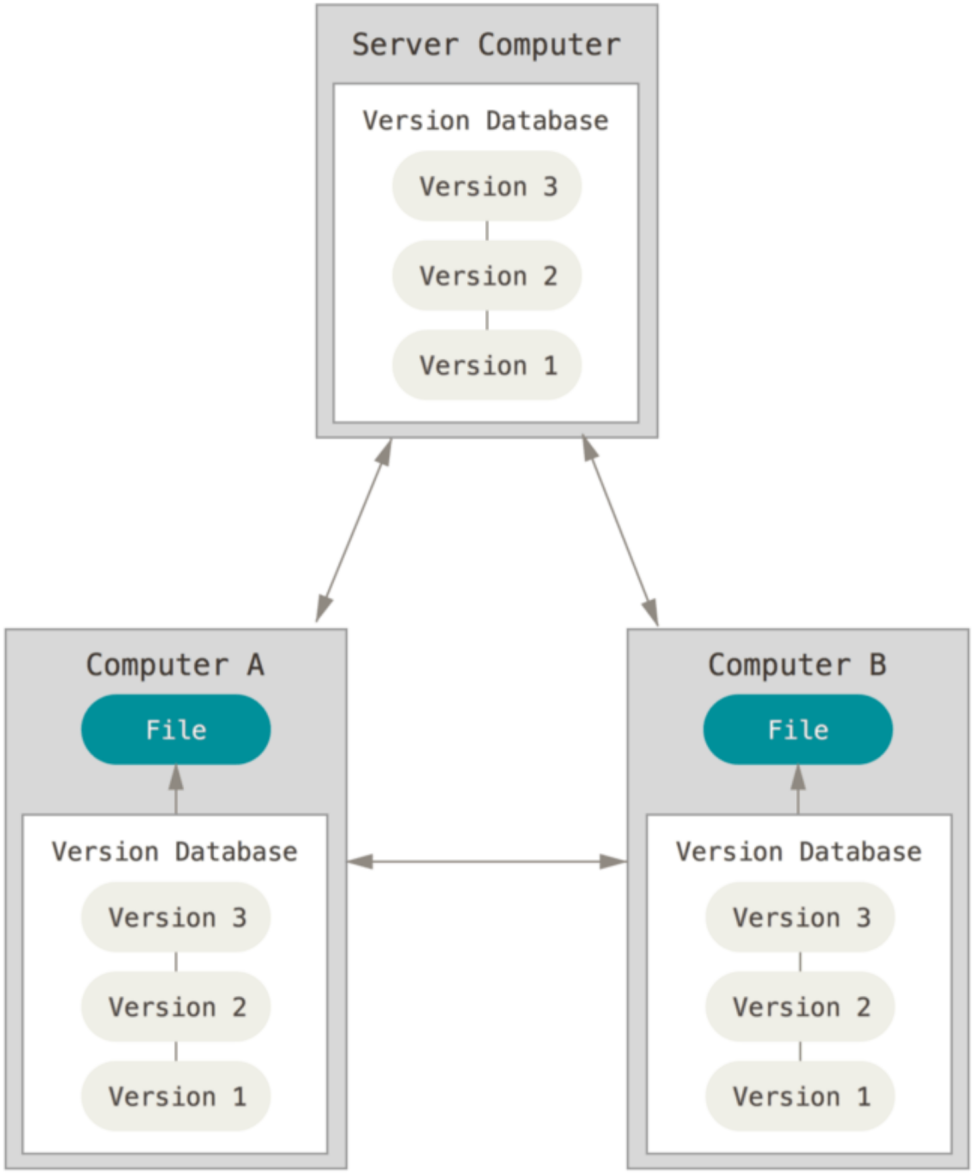
2.1 Distributed Version Control System

As you probably know, GitHub is a Version Control System (VCS) – it stores in repositories (often called repos) all the versions that you commited from your project, so you don't lose your project's history. However, a key feature of GitHub that differentiates it from older VCSs is that it is a Distributed VCS. This means that GitHub not only stores a version of your project on its cloud service (called remote repo), but also every user from a project stores the entire history of that project on their computers. So besides storing the current version of the project that you are working on (called the working directory), you also store a Git repository on your computer with the project's history (called the local repo).

This allows GitHub to be (i) incredibly fast, since it doesn't have to access the internet to access a project's history; (ii) independent from your internet connection, for the same reason, and (iii) very safe, since every user serves essentially as a backup of the project's history.

In figure 1 below, the Version Database is what we call the "remote repo." The Files in Computer A and in the Computer B are the "working directories" and the Version Database in each computer are the "local repos." Local repos, or .git repos, are by default hidden in your computer, so you'll only see your working directory. From this point onward we will see how to create local and remote repos to store your project's history.

Figure 1: Distributed Version Control System



2.2 Git Setup

- First step: install the program. You can find very simple instructions here.
- Second step: create an account on GitHub
- Third step (optional): setup your identity. Go to your terminal and type the following commands:

  $git config –global user.name ”John Doe”

  $git config –global user.email johndoe@example.com

  Don't actually copy the $ symbol, this is standard notation to let you know this command should be run in the terminal. If you need any help to understand a command, you can type the git help <verb>. For instance:

$git help config

3 Joining a Project: Cloning an Existing Repo

This section guides you through an example on how to join an existing project, probably that your Professor, or co-author, already began. This will be useful to introduce some concepts and Git commands. It will also be all that RAs will need to know 90% of the times.

3.1 Clone a repo

To clone a repository means that you are reaching out to a remote repository and copying nearly all data that the server has into a new local repo in your computer. Let's illustrate this with an example:

- First step: access the url of the remote repo.
- Second step: Open terminal
- Third step: Go to the directory where you want your repo to be:

  $ cd /Users/username/documents

- Fourth step: Go to the remote repo, click on clone or download and copy that link displayed. Then, on your terminal, clone the repository with git clone <ulr>

  $ git clone https://github.com/luigicaloi/GitShortIntro.git

This will create a GitShortIntro directory with a working copy of the latest version of the project. It also stores a .git directory (the local repo) inside it, which is hidden. For more info, see this.

3.2 Staging Modified Files

If you open the new GitShortIntro directory, you?ll see that there are only two files, ?AllReaders.tex? and "Test.do," and one folder "GitHub_a_Practical_Introduction". Your task is simple: open the "AllReaders.tex" file and include your name at the end of the list of all readers.

Now that you have made changes to the working directory, let?s see what is your status in Git. Change the directory in the terminal for the one from your new project. For instance:

$ cd /Users/username/documents/GitShortIntro

and write the following command on the terminal:

$ git status Don?t worry about the branch message, we?ll see what that is in section Git Branching. GitHub realizes that your local repo is different from your working directory, it realized that you made changes to the ?Francois_RAs.tex? file and shows it under ?Changes to be committed.? Our goal here is to make sure that your local repo also saves those changes (we?ll take care of the remote repo in the next subsection). You first need to tell GitHub what documents you want to save the changes in the local repo with the following command:

Figure 2: Git status

```
On branch master
Your branch is up to date with 'origin/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:   Francois_RAs.tex

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

$ git add Francois_RAs.tex

Now run git status again and you'll see:

Figure 3: Git status

```
On branch master
Your branch is up to date with 'origin/master'.

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

        modified:   Francois_RAs.tex

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

3.3 Committing Modified Files

Francois_RAs.tex is now under the 'Changes to be committed.' Next, we need to ?commit? those changes. Committing is telling GitHub to take a snapshot of your project in the working directory and save it, with the changes, to your local repository. In other words, you are "saving" your latest changes to the git repo. See here here for more details. The command to commit changes is:

$ git commit -m 'A message explaining your changes to the project'

3.4 Unmodifying a Modified Files

If you accidentally modified a file, or regretted the changes, and wants to revert it back to how it was when you last committed (the local repo's last snapshot), you can run the following command:

$git checkout – <file>

However, be careful, the changes you had made and not committed will be completely deleted.

3.5 Pushing to Your Remotes

So you have already (i) cloned all the data from the remote repo and created a local repo; (ii) modified your working directory and (iii) added and committed your changes from your working directory to your local repo (the .git repository). We'll now update your changes to the remote repo. This can be done with the following command:

$git push origin master

Or more generally

$git push <remote> <branch>

'origin' is the automatic name that GitHub assigns to the remote repository. 'master' is the branch in the remote repo, but don't worry about it yet, it will make sense after we see Git

Branching. Git push uploads the changes from your local repo to your remote repo. Now your working directory, your local repo and your remote repo are all on the same stage.

4 Creating a Repo

4.1 Creating an Empty Repo

In the last section we saw how to clone a repo, make changes to your local repo and push it back to the remote. In this section we will simply see how to create an empty remote repo, then you can follow the workflow from Section 3 to add your files to it.

- First step: Go to GitHub's website and login.
- Second step: On any page, click the + button on the upper right part of the screen and then click New repository
- Third step: Choose the name of your repo and click Create repository.

For more options, see this

4.2 Adding an Existing Project to Your New Repo

To add an existing project to the new remote repo, we will first create a local repo, add the existing project to it, commit the changes and push it to the empty remote repo.

- First step: Change the current working directory to the desired local's project dir.
  $cd /Users/username/newrepo
- Second step: Initialize a local git repository in your current directory
  $git init
- Third step: Add the files for your new repo
- Fourth step: Commit the new files, which creates a snapshot of how they are now and save it to the local repo
- Fifth step: Go back to your remote repo on GitHub's website and copy the remote repository URL.
- Sixth step: Add the new remote repo from your terminal $git remote add origin <remote repository URL>
- Seventh step: Your local repo is now connected to your remote repo. Push your changes to your remote repo as we saw in Pushing to Your Remotes.

For more details and options, see this.

5 Branching and Merging

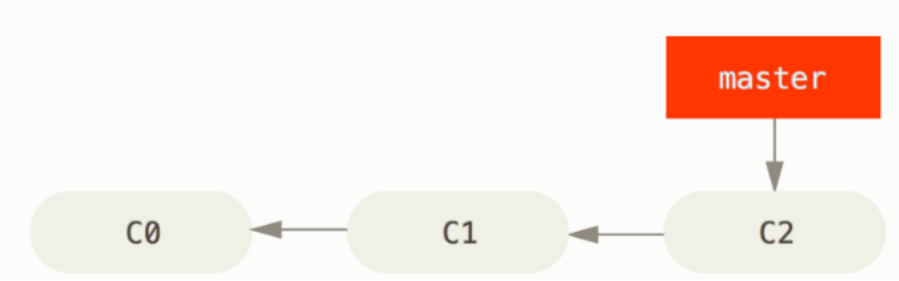
In this section we will see the two concepts that are most important for collaborative work – branching and merging.

5.1 Git Branching

The easiest way to understand branches is to think about the repo as a series of commits. When you create a repo, GitHub automatically creates a branch called master. This branch is simply a pointer to the last commit of your project. In figure 3 bellow, for instance, the master branch is pointing to the third and last commit (C2).

Now let's say that you need to make changes to your project for a new submission, but you

Figure 4: Branch Master

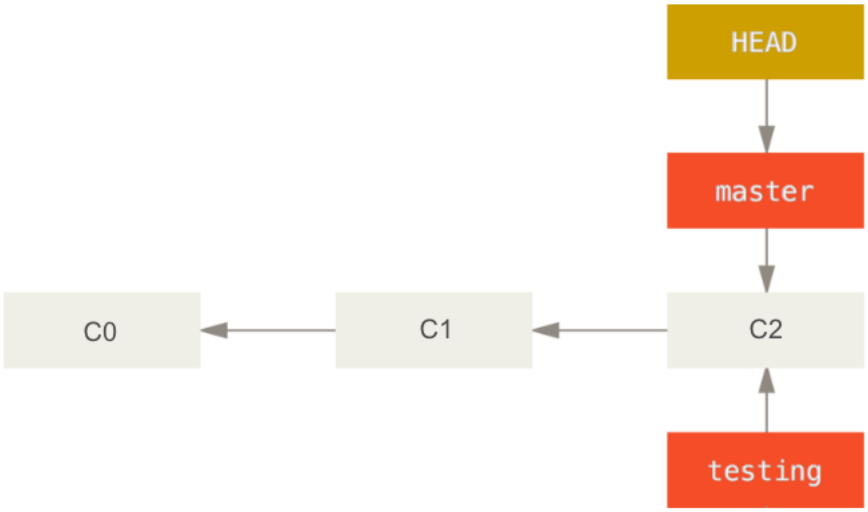


don't want to change the original code yet. You can create a new branch, called testing, with the following command

$git branch testing

Now the two branches are pointing to the last commit (see figure 5). The HEAD pointer shows

Figure 5: New Branch



which branch you are currently working in your local repo. The following command switches control to the testing branch so that you can make the new changes to your project.

$git checkout testing

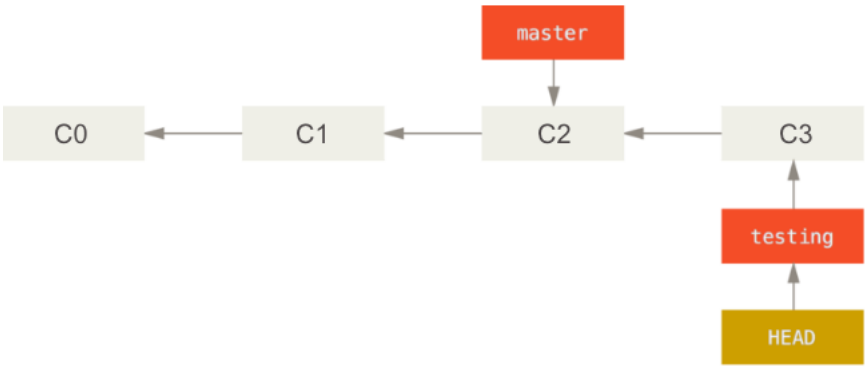
Now if you make changes to your project and commit them, your project history will look like Figure 6. Note that the master branch is still pointing to commit C2. The good thing is that

if we decide that the new ideas were bad, we can simply go back to the master branch with

$git checkout master

With this, all the files on your working directory go back to how they were when you made the last push in your master branch (C2).

Figure 6: New Branch

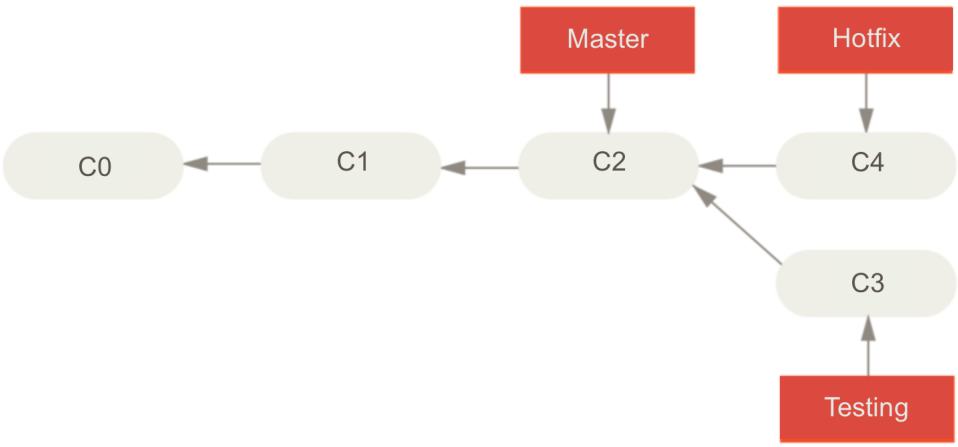


Another scenario that illustrates why it is good to make new changes in branches is when a journal asks you to make a change some details to a paper in which you've been making drastic changes. You can easily go back to your 'master' version (assuming this is the one you submitted to the journal), make changes to it, and then go back to your 'testing' branch to continue working on the big changes.

5.2 Git Merging

Following from last subsection's example, let's suppose that while you made changes to the 'testing' branch, someone else had to make urgent changes to the master branch. For that, this person created a 'hotfix' branch. The history of your project now looks like this: Now let's

Figure 7: New Branch



say we finished the changes in the hotfix branch and we are ready to merge the master branch with the hotfix branch. This can be done with the following command:

$git checkout master

$git merge hotfix

And we would see the following message: Because the commit C4 from the hotfix branch

Figure 8: New Branch

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

was directly ahead of C3, GitHub can simply "fast-forward" the master branch and change the pointer from C3 to C4. Since the hotfix branch and the master branch now point to the same commit, we can delete the hotfix branch:

$git branch -d hotfix

5.2.1 Merging conflicts

Things are a bit more complicated when you are trying to merge two branches that are not directly related. If some change made in the hotfix branch conflicts with changes made in the testing branch, GitHub won't be able to complete the merge and will ask your help to decide which version it should keep for that specific change. The first step would be to try merging both branches. $git merge testing

We would then check in which files there was a merge conflict with git status Any file with

Figure 9: New Branch

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

a merge conflict would be under unmerged paths. To solve the conflicts, we first need to open the files in a text editor. GitHub adds conflict resolution markers in it so that we can solve the conflict manually. The section of the file that must be solved looks like this: The

Figure 10: New Branch

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> testing:index.html
```

changes made to the master branch are under <<<<<<<< HEAD:index.html and above the ======== lines, because HEAD was pointing to the master branch (you were in the master branch). Everything bellow and above >>>>>>>> testing:index.html was in the testing branch. You can manually solve this issue by keeping the version you prefer, or a combination of both. Of course, delete the conflict signals from GitHub.

For instance, one way to solve this issue would be to substitute the entire block with this: Once you’re done solving the conflicts, the next step would be to let Git know these files are

Figure 11: New Branch

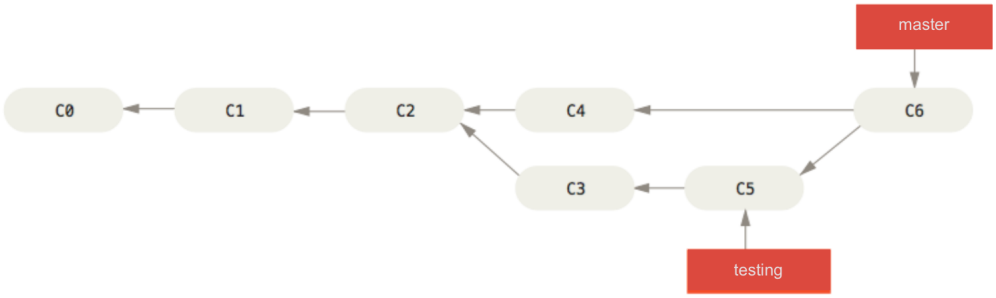
```
<div id="footer">
please contact us at email.support@github.com
</div>
```

solved by running git add in the terminal

$git add document

You can then commit your changes and finish merging the two branches. For more details, please see this. When merging branches that are not directly linked, instead of just moving the branch pointer forward, GitHub will create a new commit that includes the changes in hotfix and testing. Your new commit history will look like this: Branching and merging allow us to

Figure 12: New Branch



understand better what was going on with the push command in Pushing to Your Remotes. We saw how to push a local repository to a remote one with the command git push <remote> <branch>, but we told you to not pay attention to what was the branch part of it. Just like our local repo, the remote repo can have multiple branches (they usually follow the same structure as our local repo). The <branch> command is simply telling Git to what branch of our remote repo we want to push our changes. Second, it might have seemed like Git magically sent the data to the remote repo and made both be on the same stage. What if there are three people working on the same branch and pushing their changes to the branch in the remote repository? The push command is implicitly merging the local and remote branches. Therefore, if there are multiple people pushing to the same branch, we might need to fix merge conflicts to decide what changes to keep.

6 Using GitDesktop

While using the terminal can be very good to work on GitHub, the GitDesktop app is useful for visualizing what changed in a new push.

- First, download GitDesktop here.
- Second, add your local repository to the GitHub Desktop by going to File-¿ Add Local Repository.
- Third, On the left of your screen, choose GitShortIntro as your current repository and master as your current branch.
- Fourth, click on pull origin to make sure that your GitHub Desktop is up to date with the remote repo.
- Fifth, Under your current repository, click on "history."

This is a list of the commits to that repository to the master branch on the GitShortIntro remote repository. You can click in one of the commits to see what changes were made. In red are the deleted lines and in green that ones that were added.

Figure 13: GitHub Desktop