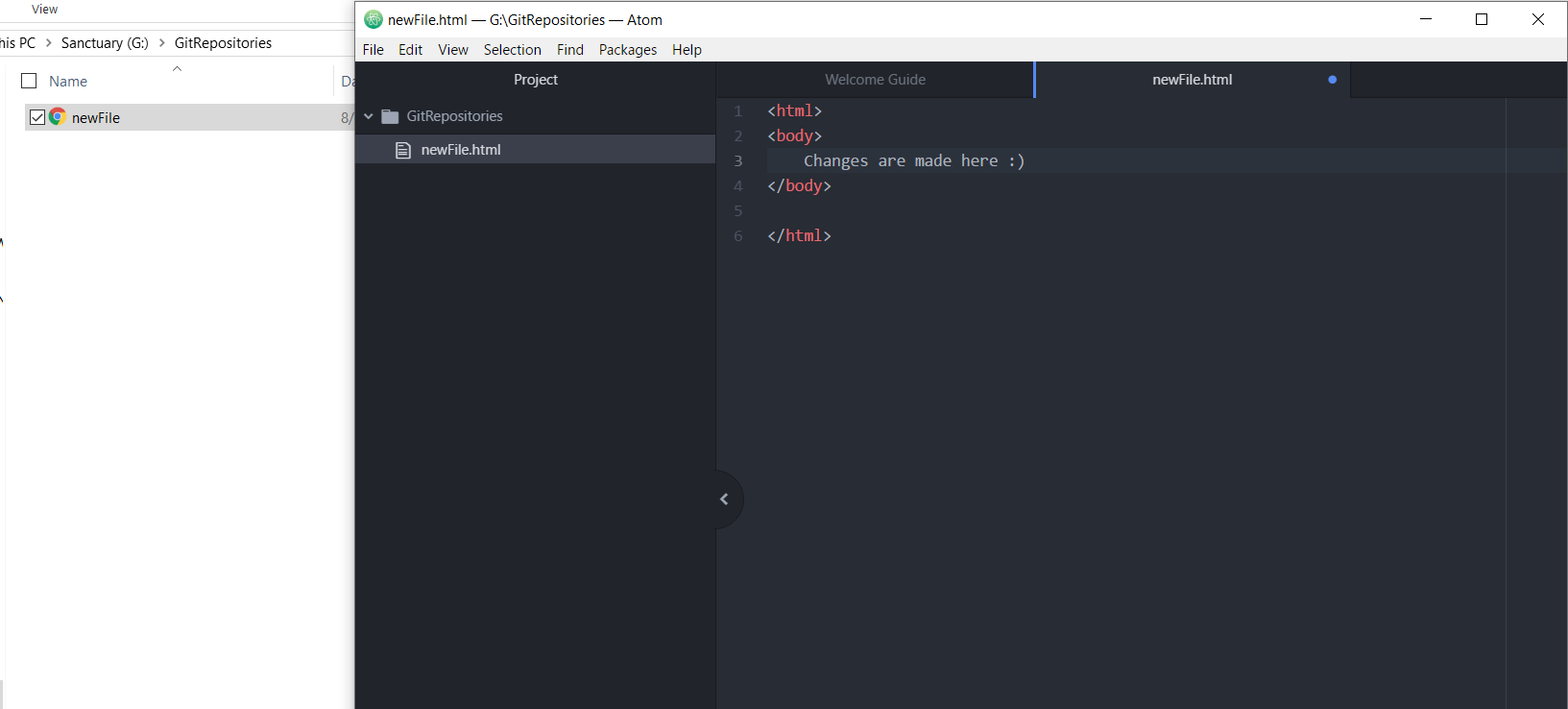
**sudo apt-get install git.**

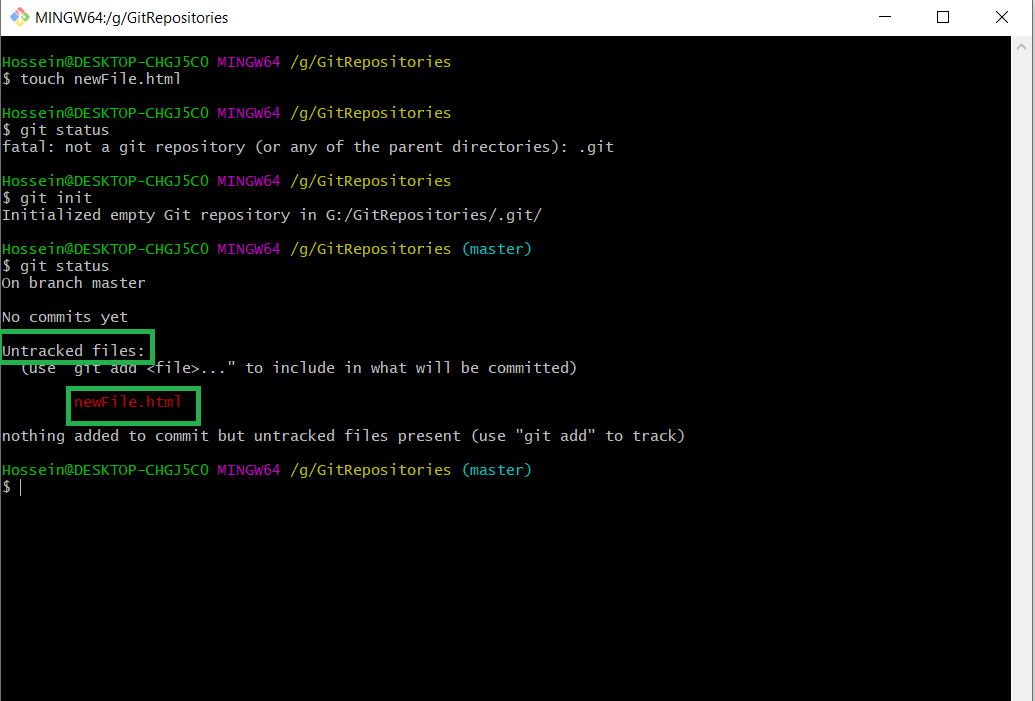
When We are in a directory and we want to start working with GIT, the first command is

$ **git init**  which means from now on this directory is under control of git.

1. make a file -> change the file -> bring it to “stage mode” -> commit it.



**$ git status**

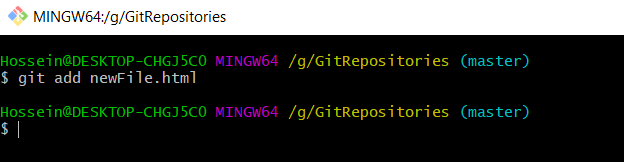
****

1. In order to make it staged, we say :

**$ git add <file name>**

**$ git add . -> adds all files that are not already added or changed somehow.**

**$ git add –A -> adds all files that are not already added or changed somehow.**



1. Let’s Tell git who we are and who is going to make changes to the file.

**$ git config --global user.email ”<emailAddress>”**

OR

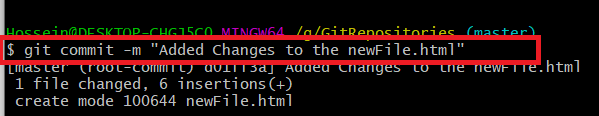
**$ git config --global user.name ”<Your name>”**

****

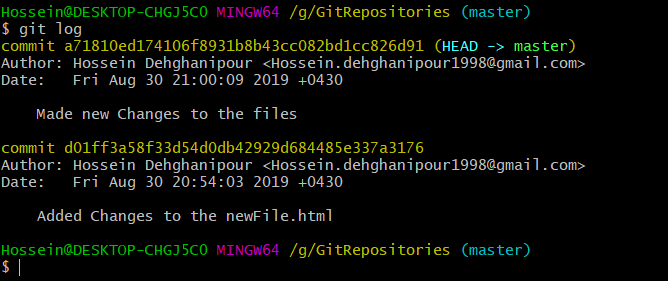
1. when we make a change, git calls it an “Untracked File|change “.

**$ git commit –m “\*”**

instead of the “\*”, we write some descriptions of what we have done (the changes) to the files.



1. **$ git log** tells us what ever we have done to this git repository and who has made what changes.



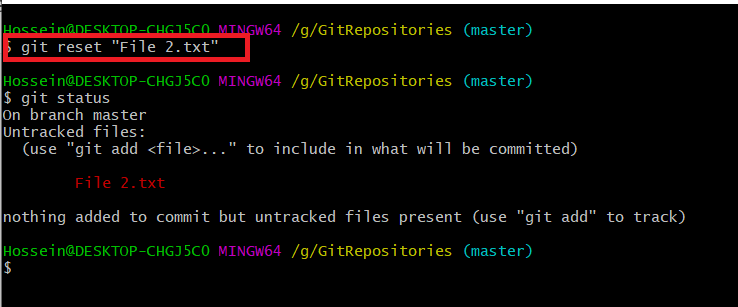
# **Head** is where we are **right now** which is usually the last repository we are working on.

1. when we make some files at the “Staged” position but we are not sure if we have made the correct changes. so we use **$ git diff --staged** to see the changes made to the files.



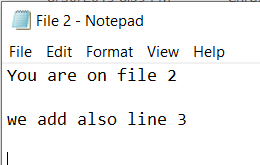
1. Assume that we have staged a file but we regret it now. there is a way to **Undo** our staging.

**$ git reset “<FileName>”**

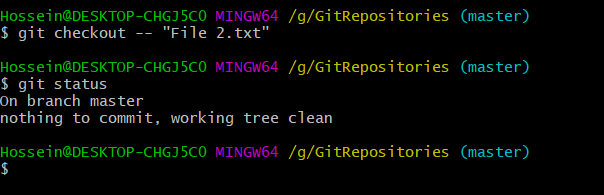


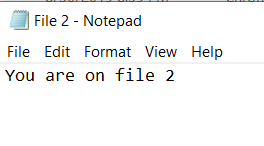
1. assume that we have made some changes to a file but we want to **Undo the Changes** and revert the changes made to the file.

**$ git checkout -- <fileName>**

 -> New changes made.

that actually brings the file to it’s **latest commit**.



 -> those new changes are discarded now.

1. we can see the changes made to the files by : **$ git diff HEAD** and also see our current head.



**Branching & Merging :**

Assume that we are working on a project and each member of the team has taken responsibility for a different part of the project. One is taking control over “Security”, one is creating the “UI” and another guy is going to create the part of our application that asks for “CAPTCHA”. So if we assume that the red line is our Master branch (Main branch), guy #1 is going to work on the CAPTCHA by making his own branch and guy #2 is also going to work on the security simultaneously and they all Merge after they have done their job. The key point in here is that they are doing different things and there might not be an overlap in their tasks.

**CAPTCHA designing**

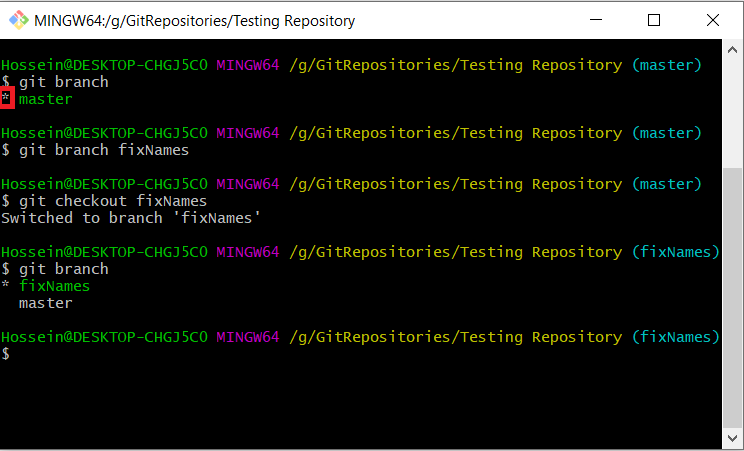
branch merge

**branch merge Master**

**Working on Security**

**$ git branch -> Shows our current branch that we are in**

**$ git branch <branch name>**

**#** you should have a commit on the master in order to create a new branch.

Our current branch

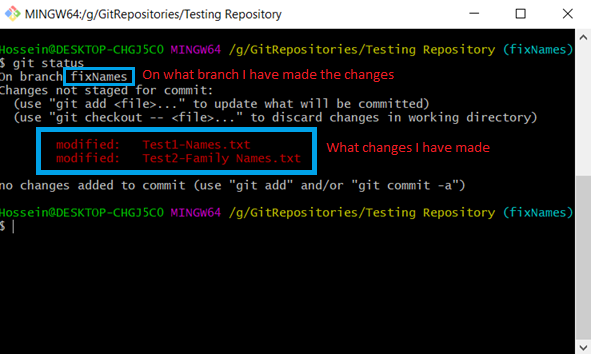
we have created a new branch named “fixNames”

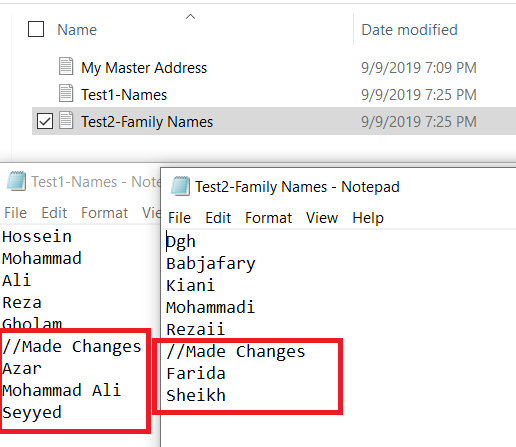
we switched from current branch to another branch

now our current branch is fixeNames after we switched

now if I add/change some files and then get the status that’s what I will see :

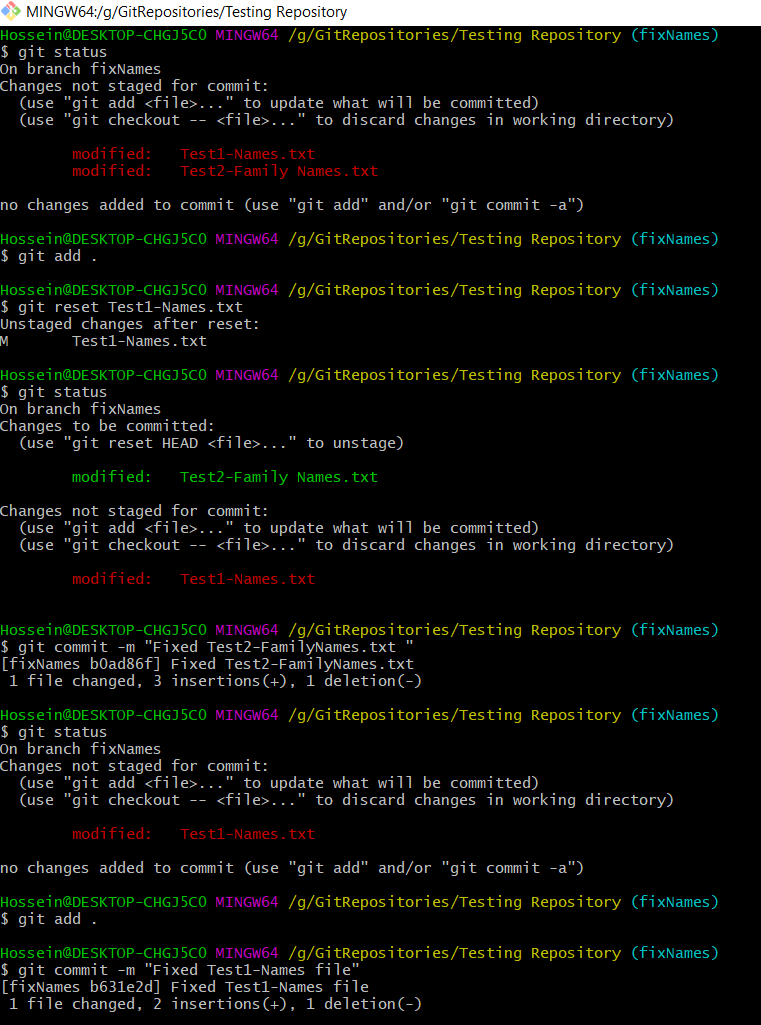


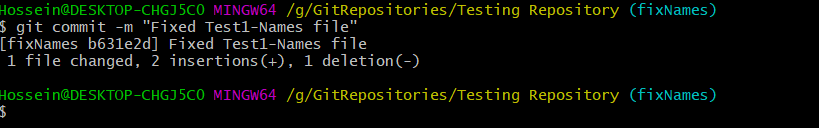




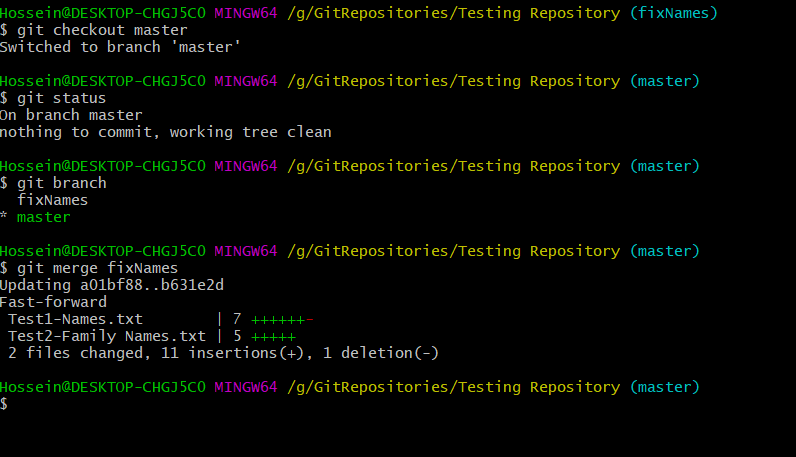
some times we make changes to multiple files but they are not in the same category. We make separate commits for different categories. so we may have already added all files. What we must do is that we “reset” the unwanted files in the current commit and then we add them to another commit.

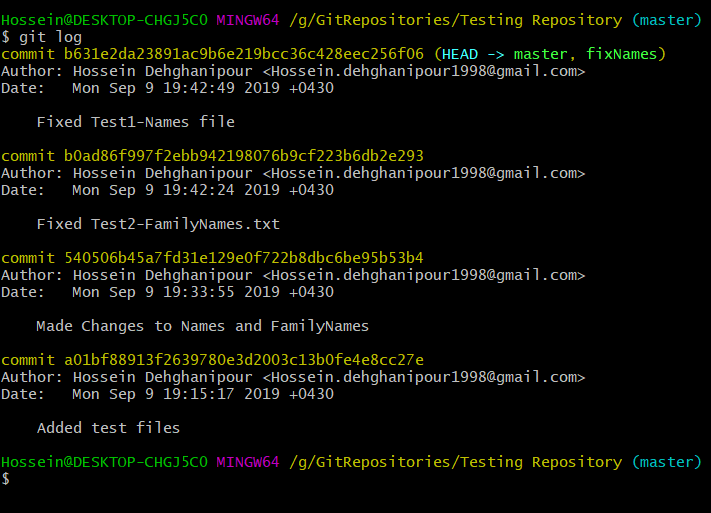
that’s where we have changed our mind





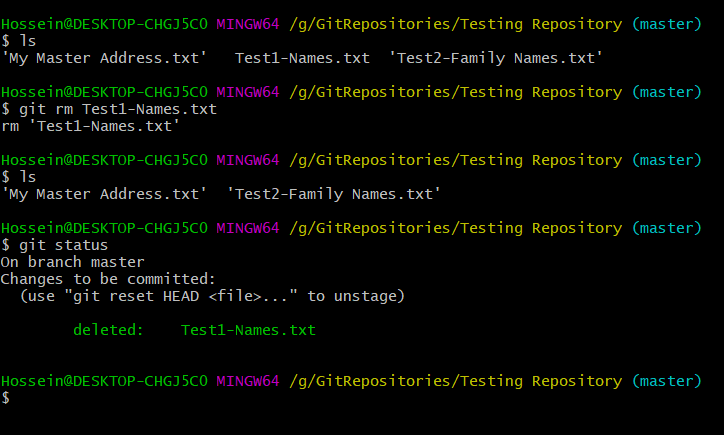
Now assume that we switch to our master branch. as we know we have made some changes in our other branch “fixNames”. In order to have the changes on out master branch too, we have to merge them :



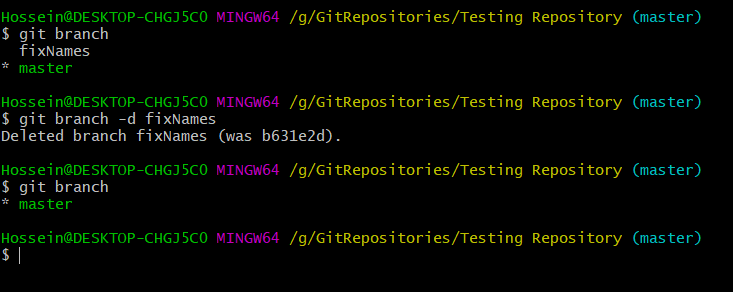


If we want to delete a file from git, we write **$ git rm <file name>** this command deletes the following file from both git and file system.

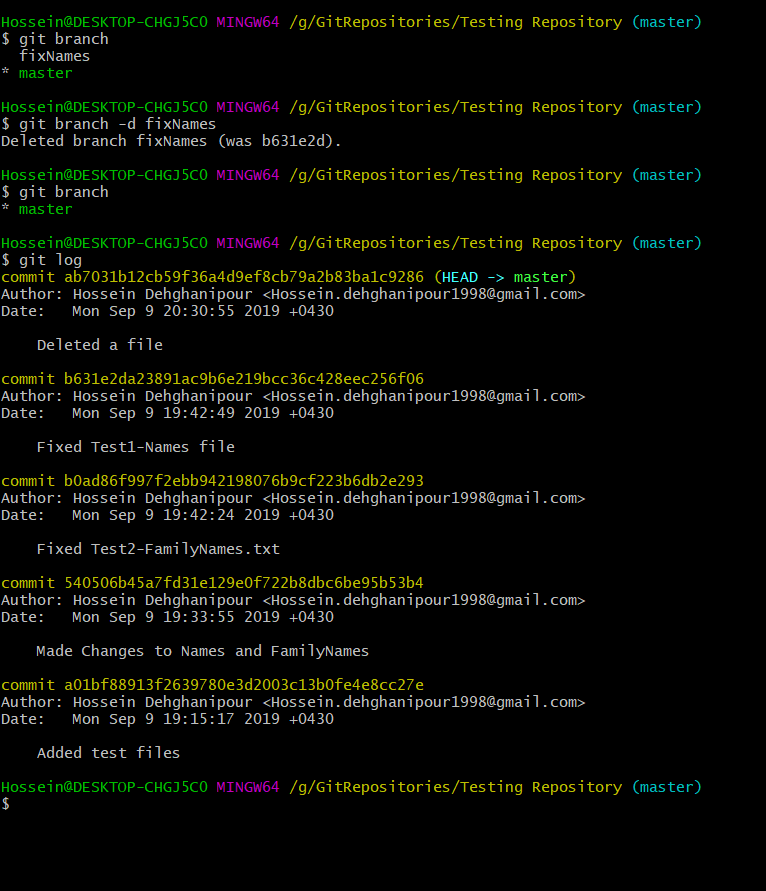
I can also use **LS** command in order to see the files in my branch.



after we have done with our branch and we have merged it, we can delete the branch. so we use the command **$ git branch –d <branch name>.**



but it doesn’t show us that we have deleted a branch :



Remotes : Origin

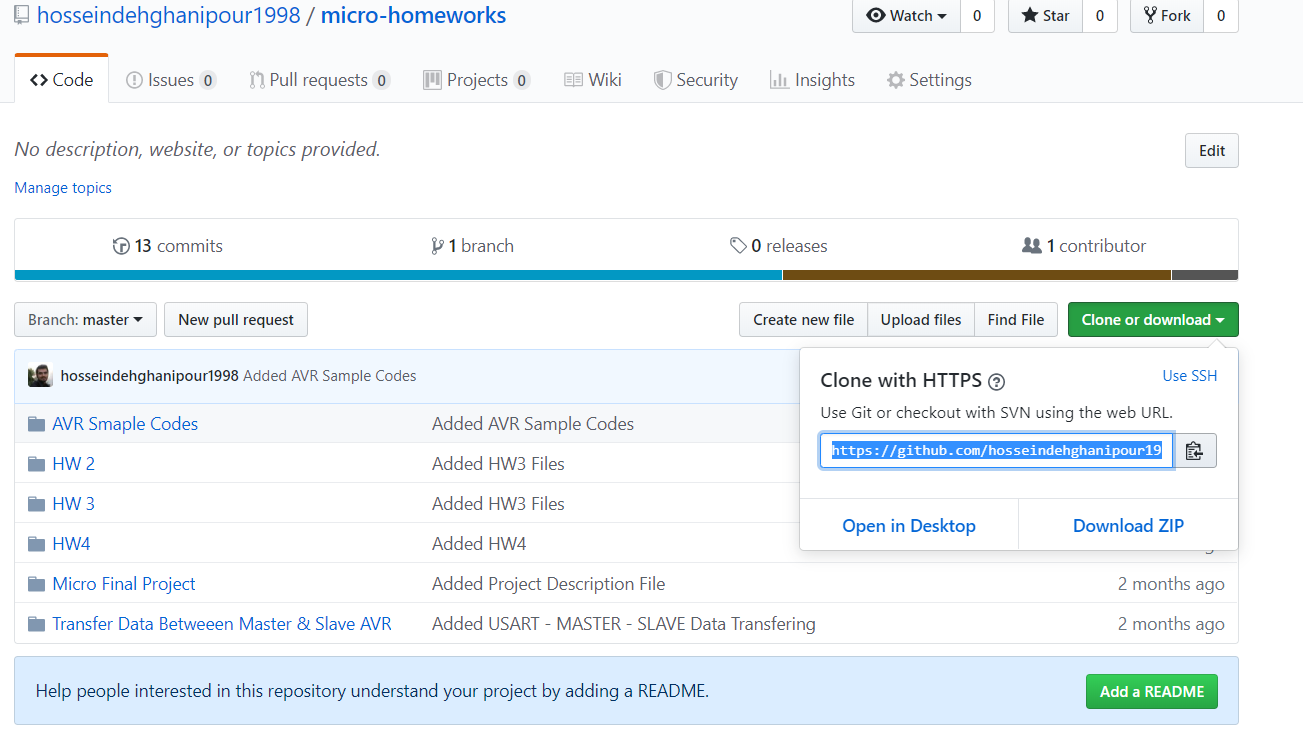
Network

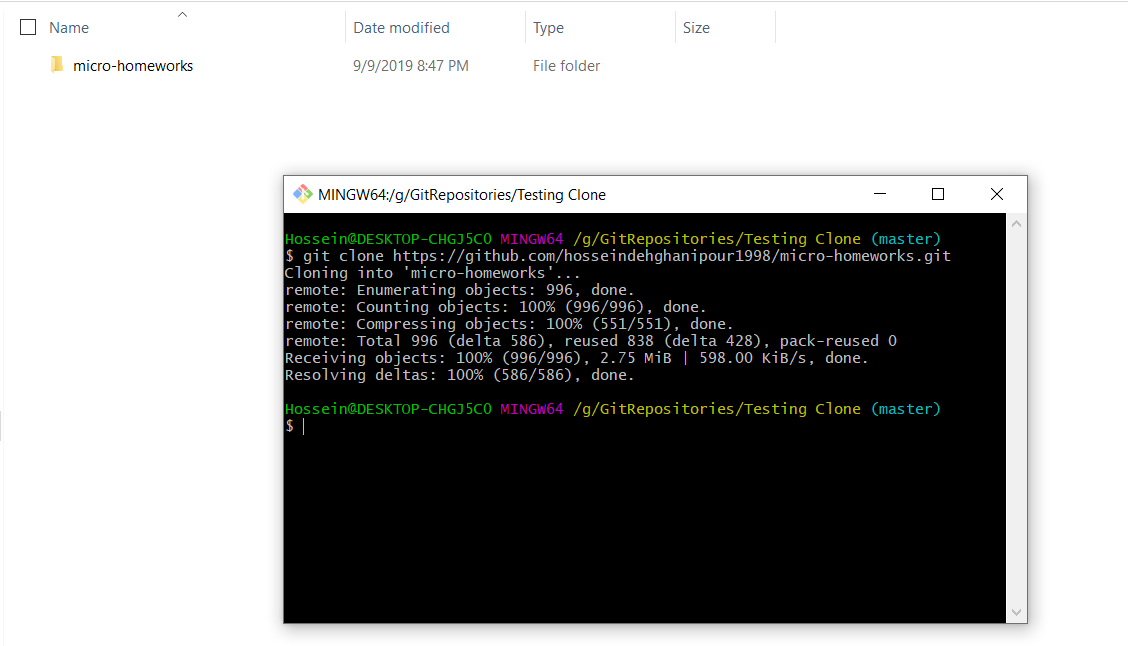
A remote Push

Pull

Clone

Clone : copies (downloads) all files from the system into your local pc.





**$ git clone <the address copied from git >**

**$ git push origin master**

when we clone a project from github, the remote set to the master of the project is usually “origin”.

**$ git pull origin master**

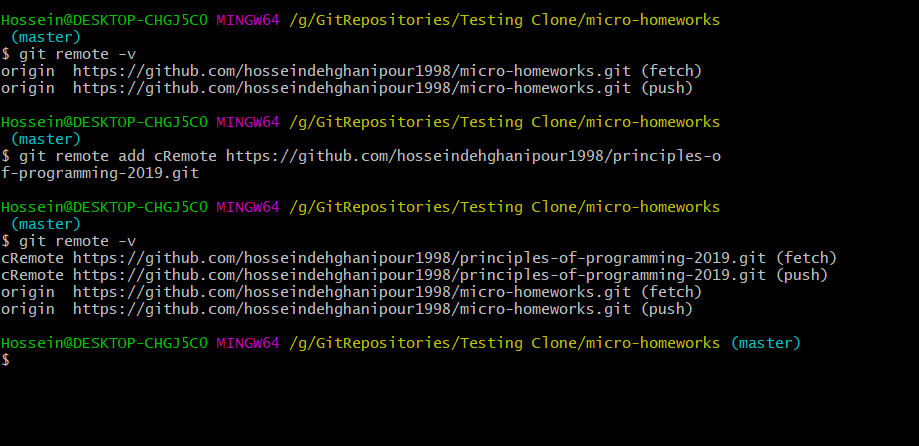
# if we have a public repository, we can pull from it but pushing to a repository requires the owner’s permission.

# we can add a remote to our directory. we can make remote to any directory from any other directory. If the directory doesn’t exist, we will face an error that tells us the path/URL we are aiming for doesn’t exist as a GIT repository.

# our projects can have more than one remotes. We can push our project onto two different servers or pull from two different servers. for example, as a new commit is made we can push it to both gitlab and github simultaneously with two different remotes pointing to different servers.

**$ git remote add <URL>** -> creates a new remote to the porject

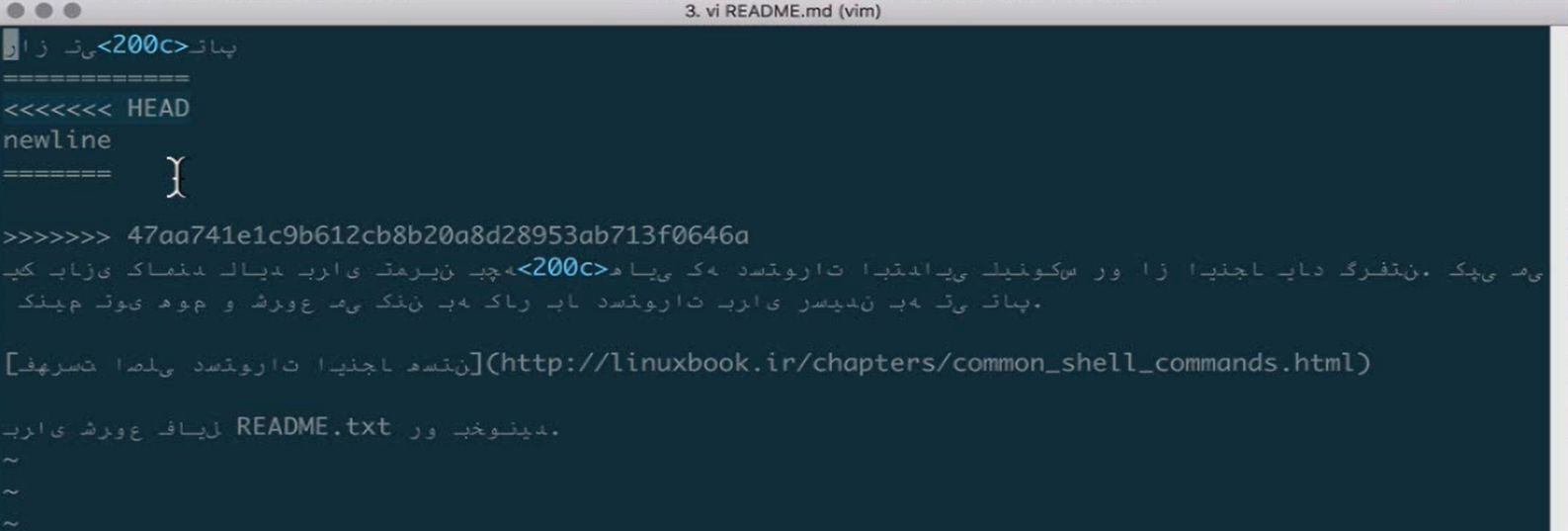
**$ git remote –v** -> shows us all the available remotes in the project**.**

****

In the illustration above we have created another remote to another repository of github named “ Principles-of-programming-2019 “ on which we can push and from which we can pull.

# we can insert the copied URL into the terminal by pushing **“ ctrl + insert “**

# some times we have a major problem. This problem will be caused when two different users are making changes on one same file. They both commit on one exactly the same branch. When one of them is going to push on the branch his own changes he will face an error saying that “ error: failed to push to URL ... “.In these situations you have to pull first in order to have his changes and then push your own changes on the branch. The git would automatically merger your changes as you pull the project but something worse might happen which is both of the programmers have changed same lines of code which is the base of a major conflict in the file. git status will tell you that you both of you have made modified changes. You have to handle it manually by yourself. If you open the file you will see such a thing :



from the “<<<<<<HEAD” till “========” are the changes you have made and the line under the “=======” are the changes the other guy has made to this file.

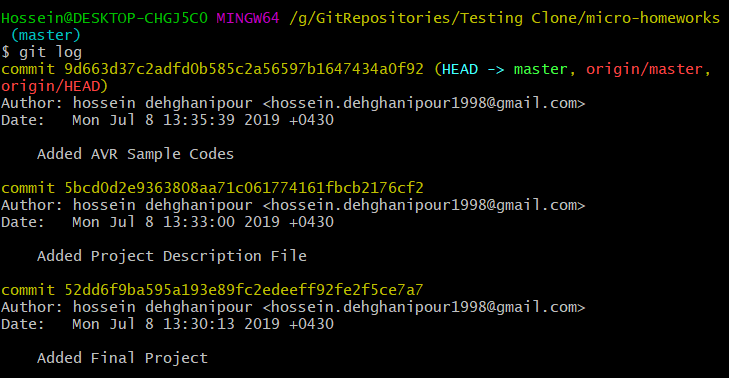
and the line “ >>>>>>>>>>47.... “ is the name(hash) of the commit.

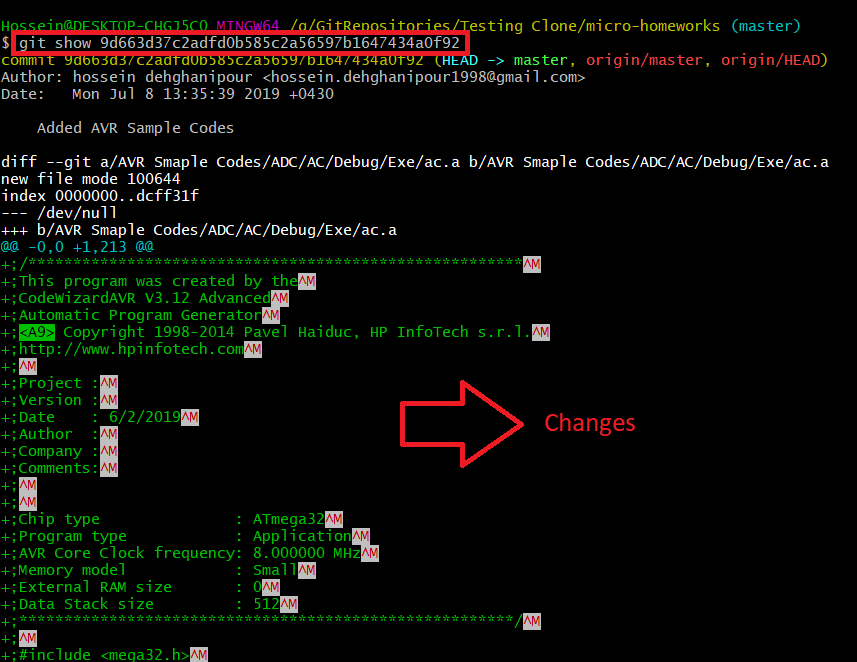
You can currently change and edit the file in order to solve the conflict and then make a new commit in order to push on the branch.

by **$ git log** you can see that you have resolved the conflict.

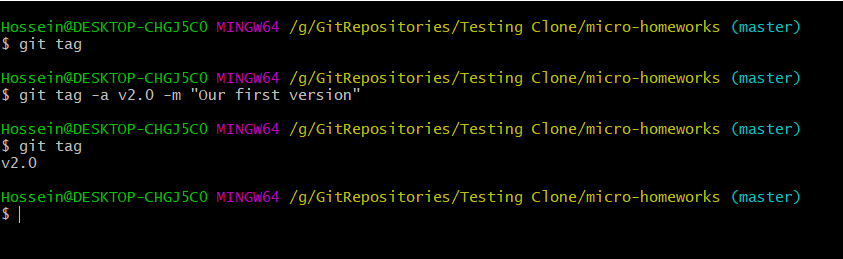
But if the both programmers change different parts of the code or file , the git will merge them together and doesn’t involve you to fix it manually.

# by entering “git log” command you can see all the commits and their commiters. but what else you can see is that it shows you the hash of the commits. by command **$ git show <commit hash>** you can see the details of that commit .

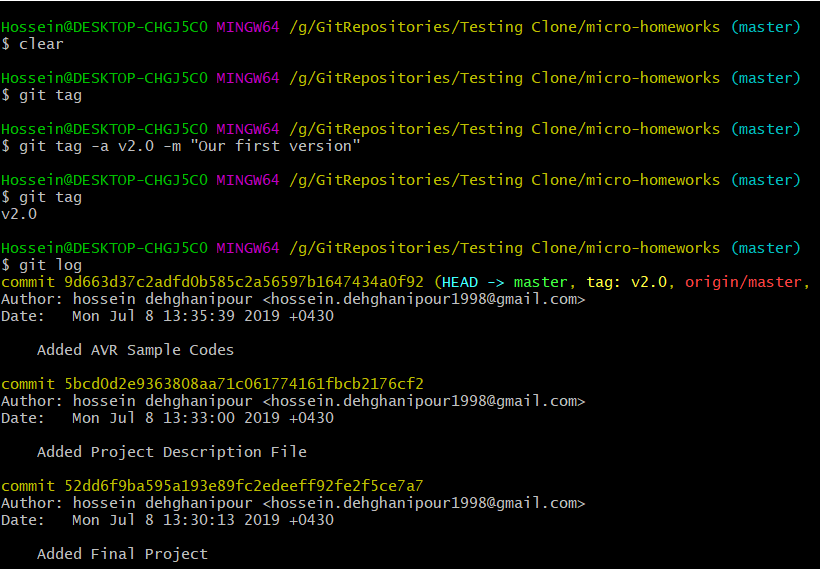
****

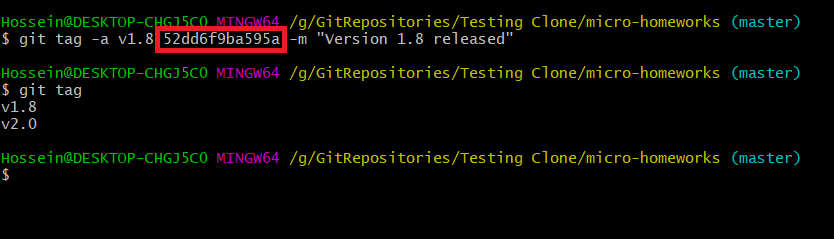
****

# After our project has reached a level that it’s suitable for usage for the client, we public a version of our application. Or when we update or improve the app we upgrade it’s version. As an example we call it “V 1.1.2”. We call this operation tagging which means that we tag a label to our commit in order to remind ourselves that this commit is the first version of our program we have released and in the possible future when we want to look for our first released version we search for the tag instead of reading all changes and the codes we have written.

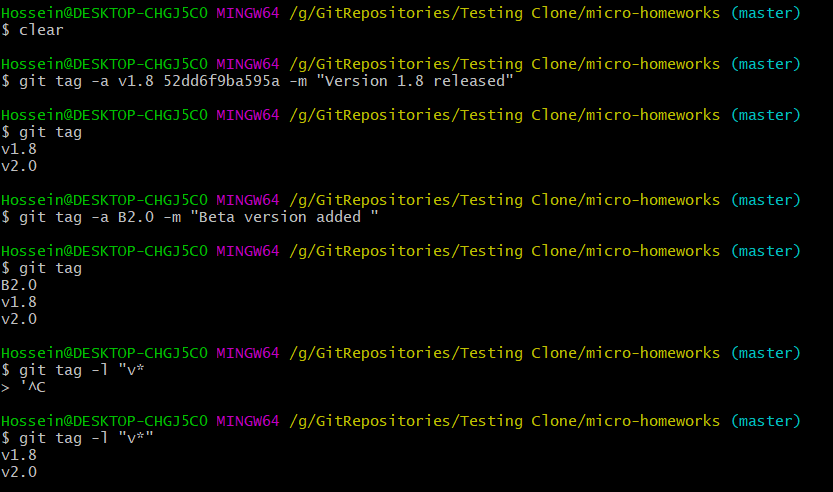


**#** notice that **–a** stands for **annotation** and **–m** stands for **Message.**

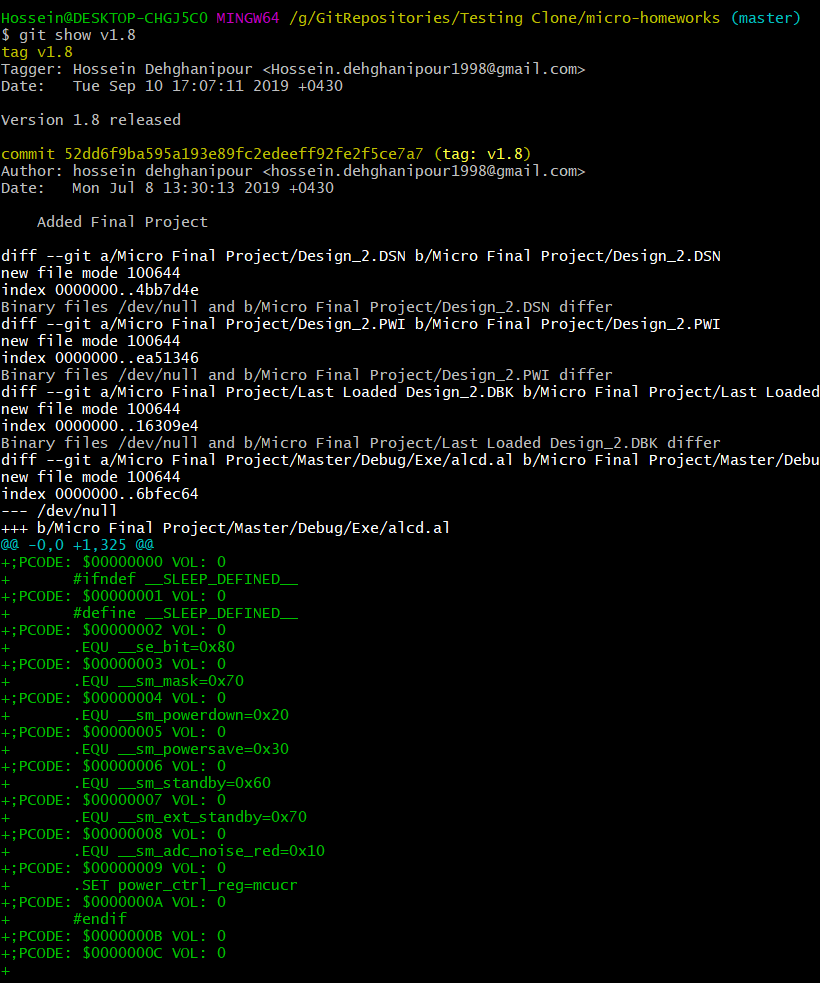
**#** assume that we are at the stage of v2.0 but we have forgotten to label our v1.8 . it’s not late at all. all we have to do is to find the commit that makes out v1.8 and copy first 10-11 alphabets of it’s hash. and then stick the wanted label to it.



**$ git tag –L “v\*” ->** shows you all of the tags that start with “v”

****

by **$ git show <version number>** we can see through the details of a specific version.



now we have made tags but they are locally made. We have to also push the tags on our origin branch.

we say : **$ git push origin --tags** which pushes the tags on the main project on the server.

I can move to version v1.8 from where I am standing right now.

**$ git checkout <version name> ex : $ git checkout v1.8**

we are now standing on branch v1.8 but we can’t make changes to this version and then commit it here. what we can do is make another branch on this version and make the changes on that branch then merge it with master.

# The hash we see as commit is something beyond a simple string. That’s a security procedure in order to avoid someone else making changes to the project in your name. So if it says that the commit is made by Hossein, That’s definitely true. If someone wants to change the project and make it looks like you have done it, then he has to steal all of your keys and uses them against you which is a hard work and somehow impossible to do. The encryption that is used nowadays in computers is **PGP** which stands for **Pretty Good Privacy.**

**GPG** is the encryption for **Genau** Systems.( Google it )

In this system, There are a pair of keys for each user ( we assume A & B ). We keep A key to ourselves in order to encrypt what ever we write ( we use it so make a digital Signature ) and we publish key B for everyone to be able to see what changes we have made. So if someone can read some data with Key B it means that we had the Key A to encrypt it.

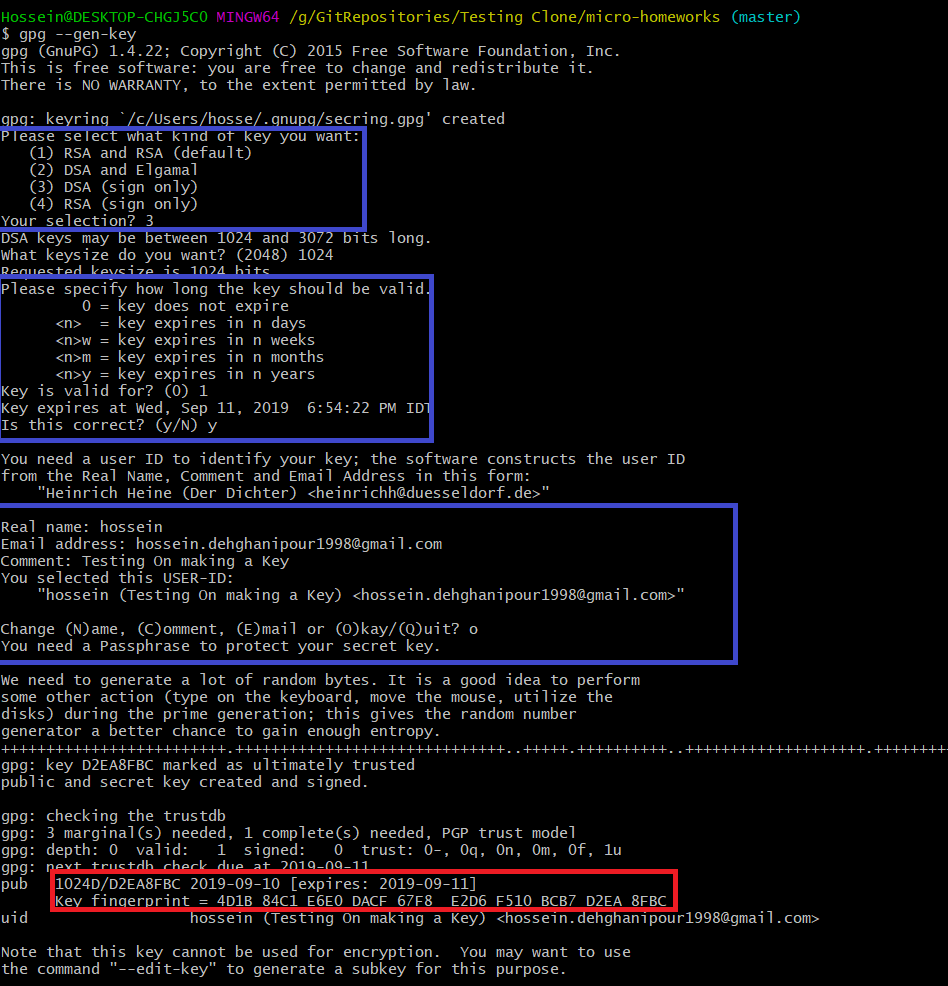
Ex : Assume there are 3 programmers. programmer A, B, C. Each one of these guys has two keys ( A1, A2, B1, B2, C1, C2 ) each user encrypts his own data ba key #1 and publishes key #2 for others to use. If programmer B uses key A2 to read some data it means that is written by Programmer A only because only his key ( A2 ) could decrypt the data.

Git works like this. Every user uses a key to encrypt and publishes the other key for others to decrypt the written data.

**$ gpg --list-keys** -> this command shows you all of your keys.

let’s make some keys :

**$ gpg --gen-key**

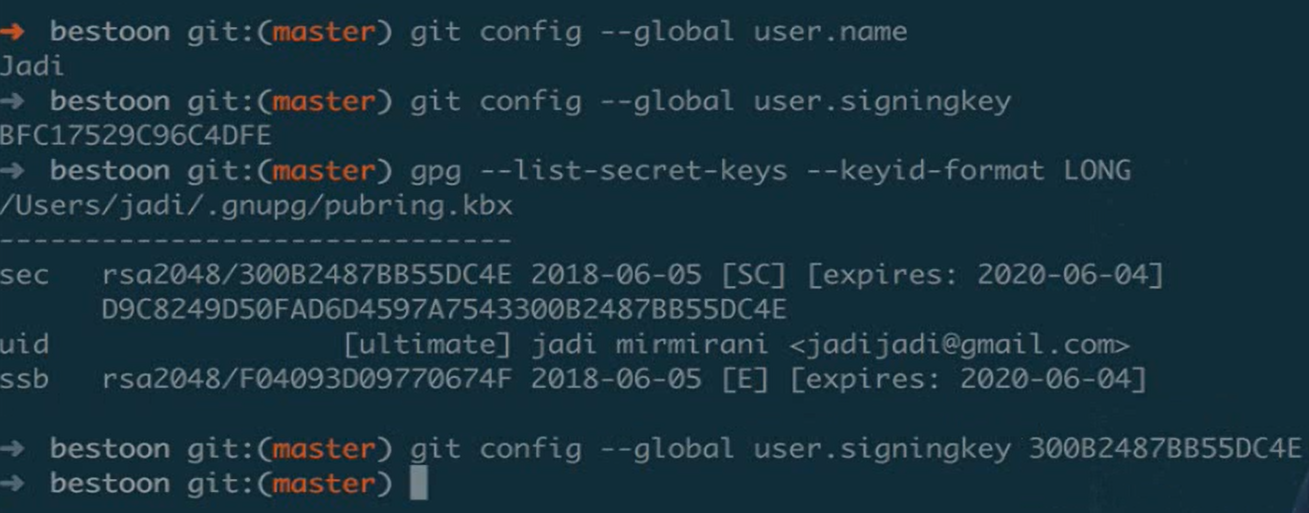
****

# the blue parts are the parts that get information from the user in order to create the key and the red part is the generated public key.

# **$ git config --global user.**signinkey this commandtells us that with what key we have signed in.

# **$git --list-secret-keys --keyid-format LONG**

By the commands below we tell git to change our sign in key to what we give it ( actually what we give must be a valid gpg key that we have already made ) :



# as we have already said we can label our commits but if we write **–s** instead of **–a** it means **Sign** ( not **annotate** ) and after this command git requires a password from us in order to make sure that we are the real us :) and by typing :

**$ git show v2.1** then it shows who has released it and shows his digital signature key.

because some times some problems might be caused, either all members of the team should sign or none of the members should sign. Be aware that if you want to sign then you should sign all of the commits you make.