

Version Control Using Git

ELECTGON
www.electgon.com
contact@electgon.com

08.04.2018



Contents

1	Introduction	4
1.1	What is Repository	4
1.2	Version Control systems	4
1.3	Topologies of Repository Systems	4
1.3.1	Centralized Version Control systems (CVCS)	4
1.3.2	Distributed Version Control Systems (DVCS)	5
1.4	Tools Used in Each Topology	5
2	Git Basics	6
2.1	Snapshot Storage	6
2.2	Staging	6
2.3	Branching	6
2.4	HEAD Pointer	6
2.5	Tagging	6
2.6	Git Commands	7
3	Git Setup	8
3.1	Download Git	8
3.2	Configurations	8
4	Building a Repository	9
4.1	Initialize or Copy	9
4.2	Cloning From Different Machine	10
5	Setting Repository for Multiple Users	11
5.1	Create Repository as Shared:	11
5.2	Create Shared Account	11
5.3	Multiple Users for Multiple Repositories	12
6	Remote Connection Using SSH	13
6.1	Generate ssh-key at Client Side	13
6.2	Add ssh-key at Server Side	14
6.3	Limiting User Activities to Git Only	14
7	Files Operations in Git	16
7.1	Files Status	16
7.2	Adding/Committing	16
7.3	List files	17
7.4	Fix Commit	17
7.5	Remove Files	17

7.6	Remove File in Old Commits	17
7.7	Ignore Files	18
7.8	Types of Allowed Files	18
8	Local and Remote Repositories Communication	19
8.1	Define Remote Repository	19
8.2	Push Changes	19
8.3	Pull Changes	20
8.4	Fetch Changes	20
9	Branches	21
9.1	View Branches	21
9.2	Create Branch	21
9.3	Switch Branch	21
9.4	Delete Branch	21
9.5	Merge Branch	22
9.6	Merge Commit	22
9.7	View Content of Branch	23
9.8	Pushing to Remote Repository	23
9.9	Track Branch	23
10	Tagging	24
11	Repository Views	25
11.1	History Review	25
11.2	Difference Review	25
11.3	Check Difference with External Tool	26
12	Submodules	28
12.1	Adding Submodule	28
12.2	Clone Submodule	28
12.3	Get Submodule Updates	28
12.4	Push Submodule Updates	29
13	GitHub	30
13.1	Basic Operations	30
13.2	Fork and Pull Request	32
14	Use Cases	36
14.1	Create Repository at Server from Client	36
14.2	Files Shared between Different Branches	37
14.3	Check Status of Specific File	40
14.4	Branch Merge Conflict	40
14.5	Merg Branches in GitHub	41
15	Summary	44

1 Introduction

1.1 What is Repository

In software development, a repository is a central file storage location. It is used by version control systems to store multiple versions of files. While a repository can be configured on a local machine for a single user, it is often stored on a server, which can be accessed by multiple users.[1]

1.2 Version Control systems

Version control is used to manage multiple versions of computer files and programs. A version control system, or VCS, provides two primary data management capabilities. It allows users to

- 1) lock files so they can only be edited by one person at a time, and
- 2) track changes to files.[2]

1.3 Topologies of Repository Systems

There are currently many version control systems in the market. Some of them are open source, others are proprietary. These tools, however, are classified into two types:

1.3.1 Centralized Version Control systems (CVCS)

First let's look at CVCS as shown in figure 1. The central repository serves as the hub and developers act as separate spokes. All work goes through the central repository. This makes version control easy and sharing difficult.

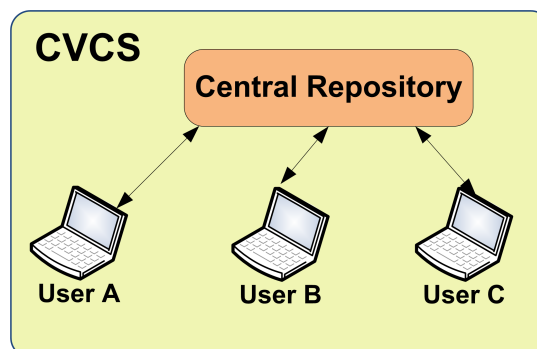


Figure 1: CVCS System

When you're working with a centralized version control system, your work flow for adding a new feature or fixing a bug in your project will usually look something like this:

- Pull down any changes other people have made from the central server.
- Make your development, and make sure they work properly.
- Commit your changes to the central server, so other programmers can see them.

1.3.2 Distributed Version Control Systems (DVCS)

With DVCS there is more interaction directly between developers as shown in figure 2. Systems are designed with the intent that one repository is as good as any other, and that merges from one repository to another are just another form of communication.

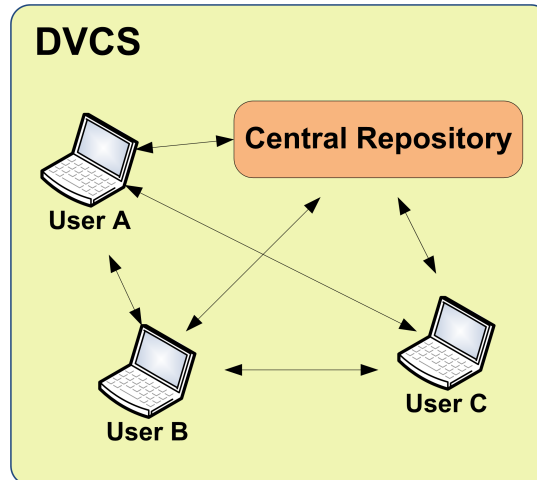


Figure 2: DVCS System

When you work with distributed version control system, your work flow will be like this:

- Cloning the original repository, you will have local repository then.
- Make your development, and make sure that they are working properly.
- Commit your changes to your local repository.
- If you want to update other repository, you push your updated repository to the target repository.

1.4 Tools Used in Each Topology

Here are most common softwares used as version control systems

CVCS: Open Source: CVS – Subversion (SVN) - Vesta

Proprietary: IBM Rational ClearCase – Perforce – Team Foundation Server.

DVCS: Open Source: Bazaar – Git – Mercurial.

Proprietary: BitKeeper – Sun Workshop.

2 Git Basics

2.1 Snapshot Storage

Git has been created and started to be used since 2005. It is storing versions of tracked files as “snapshot” of the file. This means it is storing a copy of the entire file when it is changing. Conventional storing method in other version management tools is based on storing the difference between versions of the file. In Git when user is committing new version of a file, Git is storing the entire content of the file, not the difference only. Other unchanged files, Git keeps a reference to it without copying it again. In that way a snapshot in Git means copying changed files and copy a reference (pointer) to unchanged files.

2.2 Staging

Storing files to Git repository is done in two steps, add then commit. This because Git needs to take snapshot first of the tracked files then stores it. The first step then (which is called add) is adding the files to a stage in which Git can take snapshot. Second step (which is called commit) is storing or registering the copied snapshot into the repository.

2.3 Branching

As any usual version management tool, You can create many branches in your repository. Each branch is able to store set of files. This branching enables users from distributing and organizing their design files according to functionality or customer needs or implementation methodology etc.

2.4 HEAD Pointer

Using any version management tool means that different and many files are checked out into the repository. In Git there is one pointer called HEAD is pointing to the last commit by default. User can move the HEAD pointer to switch between recent or old commits.

2.5 Tagging

User can mark special commits with a tag. This is used to easily switch to this special commit using the attached tag name. This tagging is used usually for labeling releases of the design files. Figure 3 shows how a repository looks like using these terminologies.

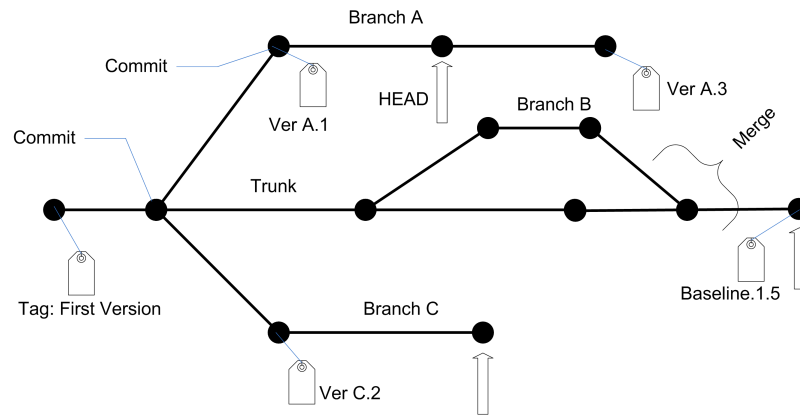


Figure 3: Repository Structure Sample

2.6 Git Commands

Git is basically a command line tool in which we use a terminal window to issue git commands. Git is using the following sample format when using it

```
$git CommandName <options> <values>
```

This means each command defined in Git shall be associated first with the word 'git' to direct the terminal to invoke git for running the commands which are case sensitive commands and can be found at this link

<https://git-scm.com/docs>

When Git is invoked, it will check first whether the current directory is a repository directory or not. This is because Git commands are used performed in a repository directory. If the directory is not a repository, Git will result in error message if you tried to execute any of its commands.

Many GUI tools exist for dealing with Git in Graphical User Interface to support different users flavors. Most and common tools used for that is GitHub which is a web-based interface used to create and handle repository actions. Not only but also some packages are installed with git to visualize the repository like gitk tool which can be invoked simply by

```
$gitk
```

Or another gui interface can be invoked by typing

```
$git gui
```

3 Git Setup

3.1 Download Git

If you are working in Windows, go to this link to start downloading Git for windows:

<http://git-scm.com/download/win>

Then start setup Git as any normal program.

If you are working with Linux, in a Terminal window then simply install Git by

```
$apt-get install git
```

Most Linux distributions come already with installed Git. You can make sure if it is installed by typing

```
$which git
```

3.2 Configurations

For launching Git in first time, you have to provide some information . This information is used by Git to track modifications done in files of your working repository. This can be set using the command

```
$git config <option> <value>
```

Also some general configurations can be done using this command like color of branches, status, etc. To see current set configuration for your Git setup, you can type

```
$git config --list
core.symlinks=false
core.autocrlf=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
pack.acksizelimit=2g
help.format=html
...
```

What is needed actually at this step is your user name and email, this can be provided by:

```
$git config --global user.name "Working Tester"
$git config --global user.email tester@test.tst
```

you can later check that your user name is set by running:

```
$git config user.name
```


4 Building a Repository

A repository in git is meta-data of different versions of the tracked files. This meta-data is usually hidden as a .git file in the directory of the created repository. Git repositories are able to track only files and folders of the repository directory i.e. it can't track files outside its directory.

4.1 Initialize or Copy

To get a Git repository, you can create it or you can copy an existing repository. To create a repository, it is advised always to think first if it will be a local repository or it will act as a central repository. Local repository means that it is aimed to include files of one developer. Of course it can be shared later with other developers but what is meant more is the user who created it is going to track his own files.

What is meant by central repository is a repository that is used by all users as a reference or as a baseline. It doesn't contain any design files, it contains only the meta-data. Knowing that Git is a Distributed Version Control system, this can cause confusion about this local and central types of repositories. To clarify it, originally in Git you can copy any repository from or to any other user. But once copied, you can't update the remote repository (that you copied from) with any changes you made in your files. You can update only a "central" repository. As a side note, you are allowed to update other remote local repository but you have to configure that remote repository.

Git is designed to work so. To make difference then between the normal local repository and the central one, when creating the repository you have to mention that. Central repository in Git is known as bare repository. So to create a bare repository you can write the command

```
$git init --bare <name_of_created_repository>
```

or you can also create a bare repository from an existing normal one by typing

```
$git clone --bare <path_to_remote_normal_repository> <name_of_copied_repository>
```

Bare repository means that this repository contains only the meta-data. No tracked files are stored in that repository. If you can browse to this bare repository in your file system, you will find it contains only meta-data. Clone means to get a copy of the remote repository and handle this copy as a bare repository.

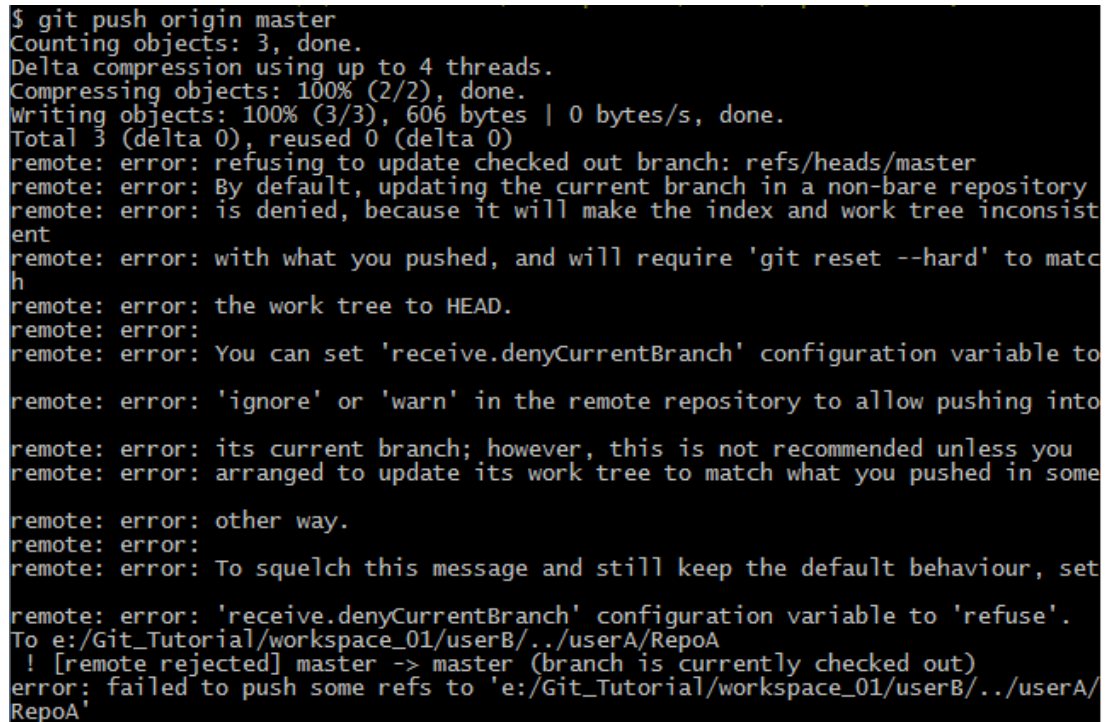
Without this mentioned switch "bare", the created repository will be handled as a normal local repository. For example, lets create a simple local repository called RepoA in a directory called userA

```
$mkdir -p userA  
$cd userA  
$git init RepoA
```

In another directory (assume it is called userB), lets create another local repository by cloning ReopA

```
$cd ..  
$mkdir -p userB  
$cd userB  
$git clone ../userA/RepoA
```

Note here that we didn't specify new name for the copied repository so that it will be copied with the same name. Now after adding some files to the copied repository, if you tried to update the original remote RepoA with these changes, it will give an error like in figure 4.

A terminal window with a black background and yellow text. It shows the output of a 'git push origin master' command. The output includes progress bars for counting, compressing, and writing objects. It then shows a series of error messages from the remote repository: 'remote: error: refusing to update checked out branch: refs/heads/master', 'remote: error: By default, updating the current branch in a non-bare repository is denied, because it will make the index and work tree inconsistent with what you pushed, and will require 'git reset --hard' to match the work tree to HEAD.', 'remote: error: You can set 'receive.denyCurrentBranch' configuration variable to either 'ignore' or 'warn' in the remote repository to allow pushing into its current branch; however, this is not recommended unless you arranged to update its work tree to match what you pushed in some other way.', 'remote: error: To squelch this message and still keep the default behaviour, set 'receive.denyCurrentBranch' configuration variable to 'refuse'.', and finally 'To e:/Git_Tutorial/workspace_01/userB/../../userA/RepoA', '! [remote rejected] master -> master (branch is currently checked out)', and 'error: failed to push some refs to 'e:/Git_Tutorial/workspace_01/userB/../../userA/RepoA''.

```
$ git push origin master  
Counting objects: 3, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 606 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
remote: error: refusing to update checked out branch: refs/heads/master  
remote: error: By default, updating the current branch in a non-bare repository  
remote: error: is denied, because it will make the index and work tree inconsis  
ent  
remote: error: with what you pushed, and will require 'git reset --hard' to matc  
h  
remote: error: the work tree to HEAD.  
remote: error:  
remote: error: You can set 'receive.denyCurrentBranch' configuration variable to  
remote: error: 'ignore' or 'warn' in the remote repository to allow pushing into  
remote: error: its current branch; however, this is not recommended unless you  
remote: error: arranged to update its work tree to match what you pushed in some  
remote: error: other way.  
remote: error:  
remote: error: To squelch this message and still keep the default behaviour, set  
remote: error: 'receive.denyCurrentBranch' configuration variable to 'refuse'.  
To e:/Git_Tutorial/workspace_01/userB/../../userA/RepoA  
! [remote rejected] master -> master (branch is currently checked out)  
error: failed to push some refs to 'e:/Git_Tutorial/workspace_01/userB/../../userA/  
RepoA'
```

Figure 4: Error in Updating Remote Repository

This error resulted as we tried to update the original repository which is not a bare one. More details about how to add files and update the repository will follow in next sections.

4.2 Cloning From Different Machine

In case of cloning from a remote machine, this can be easily done by defining the username and destination machine before the target repository path

```
$git clone username@hostmachine.com:/path/to/target/repository
```

or using the IP address of that machine

```
$git clone username@1.2.3.4:/path/to/target/repository
```

5 Setting Repository for Multiple Users

When Git repository is initialized, by default it will be initialized to a single user. Working with a team requires then to give access for team co-workers to the initialized repository in order to be able to push and pull their work files. In configuration step, we have seen how we can setup user name and email to be associated with the created repository. Git is NOT designed to have multiple user names in one repository. This means that there is no mechanism in Git to setup multiple user names in the repository. By default the user account who has created repository is the only user who can push and make changes to the repository.

In order to overcome this point and allow team co-workers to push and make changes in the repository, there are mainly two configurations can be used for that.

5.1 Create Repository as Shared:

Although you can not create multiple users for the Git repository but there is another option while creating the repository to allow access from multiple user accounts. This option is to define the repository as a shared repository. This can be set using the following command while creating the repository

```
$git init --bare --shared=group RepositoryName
```

If the Git repository already created and you would like to modify it to be shared use the following command then

```
$git config core.sharedRepository group
```

By configuring the repository to be shared to group, this means that any user account who has read and write permissions on the directory in which the repository is located will be able to push and make changes to the repository. Simply, if the Git repository is located in directory RepoDir, group members who has read and write permissions will be able to modify the repository. Because of that, you have to make sure that needed user accounts are included in the directory group. So it is always advised to define a group for Git users in your Operating System, then assign group property of RepoDir to the created Git group. This can be carried out using the following commands

```
$chgrp GitUsers /path/to/RepoDir  
$chmod g+ws /path/to/RepoDir
```

5.2 Create Shared Account

The other way to have multiple access on the repository is to share the user account who is registered in the Git repository with your team co-workers. Each member of the team will use this account then to push and pull from the repository. We can show here how we can add this user account in Linux system. For our demo here, lets create a user called ProjUser.

```
$sudo adduser ProjUser
```

then add password for this user.

Note that the way how a user account to connect to a remote git repository is discussed in next section when discussing connection using SSH.

5.3 Multiple Users for Multiple Repositories

It worth mentioning here that it is possible to create different Git repositories, each repository can be configured with its dedicated used account. In Git you can create a global user name to be used for any created Git repository in a machine. despite that, you can create a repository with a dedicated user name but not as a global one. In this case, the assigned user name will be local only to this specific Git repository. The following commands can be used to configure a local user name.

```
$git config user.name "Local User Name"  
$git config user.email "name@local.com"
```

Remember how a global user name was created as discussed in previous section with the option `-global`.

6 Remote Connection Using SSH

In previous section we have discussed main settings and preparations for having multiple users who can access the Git repository. If all team co-workers are working on the same machine, accessing the repository is a local operation then. But for a user who wants to access a remote repository, connection protocol has to be defined then. Although this discussion is not related to Git operations or configuration, however it may be useful to mention it for having clear and complete guide while working with Git.

Git allows for remote connection using http protocol or ssh protocol or special git protocol. This simply means we can clone a git repository using one of these protocols. For example you can download Linux kernel using git over http protocol

```
$git clone http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

or using git protocol

```
$git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

The case of using SSH protocol is discussed more here as it is connected more with software development work structure. To clone a Git repository over SSH protocol this can be done in similar way

```
$ git clone ssh://[UserAccount@]RemoteServer/GitProject.git
```

or shortly as

```
$ git clone [UserAccount@]RemoteServer:GitProject.git
```

In previous section we have seen example for cloning a repository using SSH protocol.

```
$git clone username@hostmachine.com:/path/to/target/repository
```

6.1 Generate ssh-key at Client Side

What worth mentioning here is in SSH protocol an ssh-key is needed while connecting from one machine to another. So if the main Git repository is located in a remote server, a client machine has to send ssh-key to the server in order to get permission from the server to access its file system. Currently ssh-key is generated automatically if the user has logged in with right credential. Note here that ssh-key is generated for the client machine i.e. not for the accessing account. To avoid confusion, The logged in account is already setup in the server machine, but still a security step has to be verified by the server in order to be sure that the user is logging from a secure machine. So this security step is achieved by generating ssh-key for the connected machine.

In case of Linux, to generate an ssh-key this can be simply done by

```
$ ssh-keygen
```

The generated key will be saved in id_rsa.pub file. You can view this key using the following command

```
$ cat ~/.ssh/id_rsa.pub
```

This id_rsa.pub file shall be sent to server administrator in order to add it in public ssh-keys file as will discussed in next steps.

6.2 Add ssh-key at Server Side

At server side, you can give permission to certain machines to access the server using these generated ssh-keys. This can be done as discussed in the following steps

Go to home directory of the user that you want to add trusted machines to his account, display content of home directory of that user, we need to review hidden folders and files,

```
$ls -al
```

If a folder called .ssh is not created yet, so create it

```
$mkdir .ssh && chmod 700 .ssh
```

we need now to create a file called Authorized_keys this file should contain all keys of the trusted machines.

```
$touch .ssh/authorized_keys && chmod 700 .ssh/authorized_keys
```

what we need to do now is to write public ssh keys of these machines in this file. Assuming that, the user has generated the ssh-key as discussed in previous steps in id_rsa.pub

```
$cat /path/to/id_rsa.pub >> ~/.ssh/authorized_keys
```

Thus, the server will allow to machines with ssh-key registered in that file to access the file system.

6.3 Limiting User Activities to Git Only

There is more restriction that can be done for the user who is accessing the remote server. We can limit this user to do only Git activities. This can be done with a limited shell tool called git-shell that comes with Git. To do so, you must first add git-shell to /etc/shells

Note: you have to be root in order to be able to edit this file.

First you have to know where is the path to git-shell

```
$which git-shell
```

it will be usually in /usr/bin/git-shell. Then go to etc directory

```
$cd /etc  
$vim shells
```

then add path of git-shell to this file. Now you can edit the shell for a user using

```
$sudo chsh GitUserName
```

this will ask for which shell you want to limit this user, enter path to the desired shell: /usr/bin/git-shell

you can test if this is working by opening another remote machine and type the following

```
$ssh GitUserName@ip.of.the.server
```

it should ask for the password of GitUserName, enter the password. You should receive error message as:

```
fatal: Interactive git shell is not enabled.  
hint: ~/git-shell-commands should exist and have read and execute access.  
Connection to 192.168.147.161 closed.
```

This means that user is not allowed to login using normal ssh protocol. He is allowed to log in only within git command (“git clone ssh:\\GitUserName@ip.of.the.server” for instance).

7 Files Operations in Git

Assuming now that we have a Git repository. Whether it is created locally or cloned from remote machine, the repository will have a hidden folder called `.git` where configurations and settings of the repository are stored in.

7.1 Files Status

To start tracking or adding files to Git repository we have to know first that Git is able to track files and folders within its directory only (directory that have `.git` folder). So Whenever there is a file in that directory Git repository prompts user about this file if the user checked repository status. The following command is used to check status of the repository

```
$git status
```

To understand this, we can do quick demo by creating a repository and checking its status by issuing the following commands

```
$git init StartRepo
$cd StartRepo
$git status
On branch master
Initial commit
nothing to commit(create/copy files and use "git add" to track)
```

If there is any new file or directory has been created in the Git repository, Git will prompt about it if we checked the status again. So lets create a file in the Git repository

```
$touch Userfile_01.txt
$git status
On branch master
Initial commit
Untracked files:   (use "git add <file>..." to include in what will be
                    committed)
    Userfile_01.txt      report.txt
nothing added to commit but untracked files present (use "git add" to track)
```

7.2 Adding/Committing

As discussed before, checking in a file in Git repository is done in two steps, add then commit. To clarify how this can be used, we can use previous quick demo. Add the created file

```
$git add Userfile_01.txt
$git commit -m 'initial commit'
[master (root-commit) 10ee409] initial commit 1 file changed, 0 insertions(+),
0 deletions(-)
create mode 100644 Userfile_01.txt
```

so `$git add` is used to add the file to the stage, `$git commit` is used to check in the file in the repository. In the case of commit a message must be mentioned about committing process which is attached using `-m` switch in previous example.

7.3 List files

In our demo so far we have checked in one text file into the Git repository. Lets assume now that we have checked in other C files. To check which files do we have currently in the repository we can use ls-files command

```
$git ls-files
Userfile_01.txt
find_frame.c
find_frame.o
frames_fn.h
load_frame.c
load_frame.o
```

7.4 Fix Commit

It can occasionally happen that user forgets to add some file to make modifications to some exiting files after the commit process has been done already. In this case you can use `-amend` switch to amend last commit.

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

This command then will commit missed files or modifications with the last commit without causing new commit to the repository.

7.5 Remove Files

Since files are added in two steps. it is possible to remove a file from the stage and also it is possible to remove it after committing. To remove from the stage we can use *reset* option.

```
$git reset load_frame.o
```

You can remove file from the repository using `rm` command. But you have to be careful when using it. Using `git rm` will delete the target file from the repository and from the directory in your file system. In order to remove a file from the repository without deleting it from the system use `-cached` switch

```
$git rm --cached find_frame.o
```

7.6 Remove File in Old Commits

Previous command will remove file from last commit. But what if you need to remove the file from all old commits. The following command can be used for that

```
$git filter-branch --force --index-filter "git rm --cached --ignore-unmatch
path/to/file" --prune-empty --tag-name-filter cat -- --all
```

This command will remove the file from all commits in which it exists. If you need to update a remote repository with this prune step

```
$git push origin --force --all
```

7.7 Ignore Files

Usually during any software project development cycles, there are some files that are not needed to be added to the repository. For example compiled code (.o files) in previous example are not needed. So to direct Git repository not to include these files we have to create a file called .gitignore in which we can list all files that we don't need to commit to the repository. In our example here we can create this .gitignore file then writing in this file which files we need to ignore. Then, we commit this .gitignore file to the repository.

```
$touch .gitignore
```

Then add your files list. In our example here I have included the following files to be ignored

```
$cat .gitignore
*.o
report.txt
*~
```

which means any file ending with .o shall be ignored. Also the file report.txt shall be ignored. Also any file ending with ~ shall be ignored. Afterward, we have to commit this .gitignore file.

```
$git add .gitignore
$git commit -m "Adding Ignore File"
```

7.8 Types of Allowed Files

It is possible to store in git repository any type of files. In general we can classify files as Binary files or Plain Text files. Binary files are those that can't be read by human. It contains only binary data that are understood by running machine. This includes executable files (.exe - .elf - .bin), image files (.svf - .png - .psd), audio files (.mp3 - .wmv), compiled files (.o), compressed files (.tar - .zip), etc. Plain Text files are files that can be read by human as it is based on ASCII characters. So we can call it ASCII files. For example text files (.txt), PDF files (.pdf), Word processing files (.doc - .odt - .rtf), Programming files (.vb - .java - .c), etc.

8 Local and Remote Repositories Communication

When working in a team, design files shall be updated among other team members (if needed) or at least a central repository shall have consolidated design files. This means design files created in a local repository shall be transferred to another remote repository. Whether the remote repository is the one we cloned our repository or not, stored files in Git can be propagated using mainly push and pull commands which will be discussed here.

8.1 Define Remote Repository

If you have cloned a repository from a remote repository, you can check some details about this remote repository using remote command

```
$git remote  
origin
```

Here, the default name set for the remote repository is “origin”. If you want to know where this repository is located, you can use the command with -v option

```
$git remote -v  
origin  user_ex@192.168.178.87:C:/RepoC (fetch)  
origin  user_ex@192.168.178.87:C:/RepoC (push)
```

It is not necessary to have omni-remote repository, we can have multiple repositories. This can be done by adding needed remote repositories as follows

```
$git remote add DevA /path/to/remote/RepoA  
$git remote -v  
DevA    /path/to/remote/RepoA (fetch)  
DevA    /path/to/remote/RepoA (push)  
origin  user_ex@192.168.178.87:C:/RepoC (fetch)  
origin  user_ex@192.168.178.87:C:/RepoC (push)
```

Provided path shall be any valid path and it is not necessary to write a path to a valid repository as this will not affect any operation here. You can use later defined name (DevA for instance) to push and pull files using this name.

8.2 Push Changes

After adding changes and make necessary modifications in design files, the developer now needs to push changes that he made to the remote repository. This can be done using *git push* command. Pushing action takes place after committing the changes to local repository.

```
$git clone path/to/remote/bare/repo  
$cd repo  
...  
.. user makes modifications  
...  
$git add .  
$git commit -m 'Developer Modifications'  
$git push origin master
```

Last command has been issued as *git push origin master* to direct Git to push changes to a remote repository called *origin* and add these changes to branch *master*.

It worth mentioning here that if the remote repository has been modified before developer pushing his changes back again, the remote repository will not accept any pushing operation from the developer until he pulls new modifications to his local repository first.

8.3 Pull Changes

If remote repository from which we cloned a local repository has been modified (new files have been committed for instance), we can update the local repository using pull command

```
$git pull origin
```

We can use simply *git pull* without specifying name of remote repository, in this case Git will use default name which is *origin*. This pull command actually is getting last modifications in remote repository and merges it with local repository which means it will overwrite existing files in the same name. If some files were deleted in the remote repository, it will not be deleted from the local repository as *git pull* is doing merge for the changes. So files with same name will be affected as it will be overwritten. Therefore developer has to be careful when using this command and make sure if there are modifications in the remote repository:

```
$git remote -v update  
$git diff origin/master
```

In this command we have first gotten updates from the remote repository, then we checked the difference using diff command.

8.4 Fetch Changes

Because it is important to check changes in remote repository, There is one command in Git which we can use to get changes first without merging with local repository. In previous section we have seen it using *git remote update*. It can be done also using *fetch* command.

```
$git fetch  
$git status  
On branch master Your branch is behind 'origin/master' by 3 commits, and can be  
fast-forwarded.  
(use "git pull" to update your local branch)  
nothing added to commit, working directory clean
```

In this example, there were 3 commits difference between local and remote repositories.

To know what changes have been made, you can use *git diff* command as shown previously. If the changes are accepted from your side, you then update your local repository using merge command

```
$git diff origin/master  
$git merge origin/master
```

9 Branches

9.1 View Branches

Branching system in Git is quite simple. First you may want to check what branches currently exist in the repository, you can do that using branch command

```
$git branch
* master
```

This command will show you current branches in the local repository. If you want to check branches that are currently in remote repository, you may use -r switch

```
$git branch -r
```

By default, when you create new repository a default branch named as *master* is created and this will be the starting or main branch.

9.2 Create Branch

Lets create new branch in our Git repository by issuing the following command

```
$git branch Cdesign
```

This will create new branch called Cdesign. Note that it will create only the branch without switching to it. Now lets create another branch and see what are the branches do we have so far.

```
$git branch Pydesign
$git branch
  Cdesign
  Pydesign
* master
```

Note an asterisk is marked beside the master branch to indicate that the Head pointer is still pointing at this branch, which means any coming commit process will be attached to that branch as long we keep the header there, no matter how many branches do we have.

9.3 Switch Branch

To activate (or to make Head points to) other branch, use the following command

```
$git checkout Cdesign
```

This will switch the repository to point to our new created branch. Which means any new commit process will be attached to that branch.

9.4 Delete Branch

To delete a branch use the following command

```
$git branch -d Pydesign
```

where Pydesign is a branch name created previously.

9.5 Merge Branch

In our example so far we have two branches: master and Cdesign. Lets add one file in the master branch

```
$git checkout master
$git add hello.c
$git commit -m "adding hello world example"
```

Lets now switch to Cdesign branch and add the same file

```
$git checkout Cdesign
$git add hello.c
$git commit -m "adding hello world example"
```

Lets add more C files in that branch

```
$git add find_frame.c load_frame.c
$git commit -m "adding frame files"
```

The repository will have now a structure like what depicted in figure 5

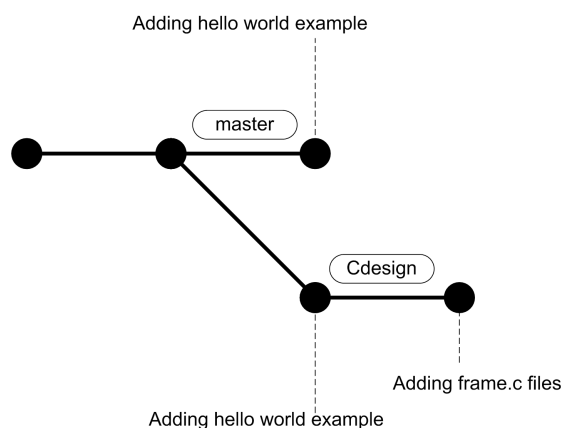


Figure 5: Initial Branch Structure Example

Note that in this example both branches have a file called hello.c. Now it is possible to merge branches in git using *git merge* command. Merging means to add contents of other branch to the current branch. For example we can merge branch Cdesign into master branch as follows

```
$git checkout master
$git merge Cdesign
```

The content of Cdesign will be in master branch now.

9.6 Merge Commit

It can happen that you don't need to merge the whole branch but you need to merge specific commit of that branch, in this case we can use "cherry-pick" command to merge one specific commit. In previous example lets assume that when we added "hello.c" file that it has been

committed with commit number “6e4592ffb32”. To merge only the changes of this commit number to our master branch, we can simply do the following

```
$git checkout master
$git cherry-pick 6e4592ffb32
```

9.7 View Content of Branch

You can have a look over what files are included in the branch using the following command

```
$git ls-tree -r --name-only branch_name
```

9.8 Pushing to Remote Repository

After finishing your modifications, you can push it to the remote repository by specifying branch name as follows

```
$git push origin Cdesign
```

And to push all branches you can use the following

```
$git push origin --all
```

Here we will update remote repository with the branch Cdesign only. While pulling you can specify also branch name to pull it only

```
$git pull remote_repo branch_name
```

9.9 Track Branch

It can happen that remote repository has some branches that are not created in the local repository. If the developer wants to update his branches to be exactly like what it is in the remote repository, he can use tracking property. Tracking originally is a technique used to match between local repository and remote repository. Matching means that both branches are linked so that if you typed *git push* it will be pushed to that matching branch without mentioning remote name or branch name. The same goes also for *git pull*. By default master branch in local repository is tracking master branch in remote repository. To know what branch is tracking which branch, you can use the following command

```
$git branch -vv
```

For the case that developer wants to have updated repository with a new branch, first he has to recognize which branches already exist in the remote repository. Then he has to fetch new updates. Then use `-track` switch to create and match the new branch that he wants

```
$git ls-remote
$git fetch origin
$git checkout --track origin/Cdesign
```

This `-track` switch can be altered with more detailed option in case you want to create the new branch but with different name

```
$git checkout -b TempBranch origin/Cdeisgn
```

You can now check which local branch is tracking which remote branch

```
$git branch -vv
Cdesign      408a932 [origin/Cdesign] hello message
* TempBranch 408a932 [origin/Cdesign] hello message
master      50f6a8c [origin/master: behind 11] changes 05
```

10 Tagging

Tagging is used in git as a mark for commits. We can make use of it to name versions for our design files. To attach a tag simply type

```
$git tag -a v1.0 -m 'first version'
```

This will add a tag called `v1.0` to the current branch.

To view all available tags of that branch type

```
$git tag
```

Important to note that tagging needs to be mentioned when push/pull to remote repositories. For example if you have branch tagged with `v1.2`, when you finish and push your branch you have to specify flag `-tag` in order to update the remote repository with the tag, typically you can do that

```
$git push remote_rep branch_name --tag
```

or to update all branches with all its tags write

```
$git push remote_rep --tag
```

To switch between tags

```
$git checkout tag_label
```

To switch back to wherever you were before checking out

```
$git checkout -
```


11 Repository Views

11.1 History Review

Since many commits are performed in the repository, it is needed sometimes to review history of these commit actions. In Git you can review logs associated with each commit process using *git log* command. It will display all logs for all commits. You can manage displayed output. For example you can review last 3 commits using the following command

```
$git log --pretty=oneline -3
408a93282b388f8f91b35796106cb3de7759e8af hello message
202d547fe7450421fcf10fbbbfcd85e3b5819a3 new hello
0378dc9d6cc295d77970c9ac260431a6b43a4ffd Merge branch 'master' into Cdesign
```

As shown, this command list last three commits shortly in one line. The first long number is called the hash value of the commit and it is generated by Git. Second field displays attached message that was used during the commit process. You can use this hash value to switch between different commits (if you didn't add a tag for the commit. If you used a tag it is easier to switch using tag value). It is not necessary to write the whole hash value you can use lowest 5 integers for example. You can even manage how this hash value can be displayed

```
$git log --pretty=format:"%h: %s" -3 --name-only
408a932: hello message
hello.c
202d547: new hello
hello.c
0378dc9: Merge branch 'master' into Cdesign
```

In last example, we directed Git to display short hash value then the attached message. Using `--name-only` we could display which files have committed in this commit.

Using this hash value, you can switch between different commits

```
$git checkout 0378dc9
```

or you can switch back the HEAD pointer by 3 commits using

```
$git checkout HEAD~3
```

To switch back to wherever you were before checking out

```
$git checkout -
```

You can view also simple graph about all commits and branches using the following command

```
$git log --graph --oneline --decorate --all
```

11.2 Difference Review

Using *git log* we can check difference occurred in a commit using the following command for example

```
$git log -p -1 --oneline
408a932 hello message
diff --git a/hello.c b/hello.c
index 651ed60..225a8ab 100644
--- a/hello.c
+++ b/hello.c
@@ -1,5 +1,5 @@
#include <stdio.h>
void main(void) {
-printf("Hello World \n");
+printf("Test Message \n");
}
\ No newline at end of file
```

where -p denotes display patch difference, -1 to check last commit only.

It worth mentioning here that you can check the difference before committing also using *git diff*. For example, if we had changes in hello.c file, we can check the difference before adding the file to stage as follows

```
$git diff hello.c
diff --git a/hello.c b/hello.c
index 225a8ab..d02120c 100644
--- a/hello.c
+++ b/hello.c
@@ -1,5 +1,5 @@
#include <stdio.h>
void main(void) {
-printf("Test Message \n");
+printf("Test Message 2 \n");
}
\ No newline at end of file
```

If the changes were added already to stage still we can check the difference using -cached switch

```
$git diff --cached hello.c
```

11.3 Check Difference with External Tool

You can integrate a third party tool to work with Git. For example we use difference comparison tool to check difference between old commits or between some files. This can be done using the *difftool* command. For example we can use Meld to compare between a modification done in a file and the committed version of the file. First we have to make sure that the tool is supported in git. This can be done use the following command

```
$git difftool --tool-help
'git difftool --tool=<tool>' may be set to one of the following:
    vimdiff
    vimdiff2
The following tools are valid, but not currently available:
```

```
araxis
bc3
codecompare
deltawalker
diffmerge
diffuse
ecmerge
emerge
gvimdiff
gvimdiff2
kdiff3
kompare
meld
opendiff
p4merge
tkdiff
xxdiff
```

Some of the tools listed above only work **in** a windowed environment. If run **in** a terminal-only session, they will fail.

Output of this commands lists then Meld as one of the supported tools. If Meld is not set in the environment variable PATH. Then we have to tell Git where to find Meld. This can be set in the config file

```
$vim .git/config
```

The config file shall be opened. Add the following configuration in the file

```
[difftool "meld"]
    path = /path/to/meld.exe
```

Or instead of editing the file manually, you can use the following command to add the configuration

```
$git config difftool.meld.path /path/to/meld.exe
```

The tool is now ready for working with Git. To compare a single file we can use the following command

```
$git difftool --no-prompt --tool=meld hello.c
```

Meld shall be opened then to show difference between current file “hello.c” the previous version. Note that `--no-prompt` switch is used to ignore some confirmation questions from Git. Instead of comparing single file, we can compare the whole repository with the previous commit using the following command

```
$git difftool --dir-diff --tool=meld 408a932
```

where 408a932 is the hash value of a previous commit. Using same method we can compare between two old commits if we know their hash values

```
$git difftool --dir-diff --tool=meld 202d547 408a932
```

12 Submodules

Some users may need to import some repositories as sub-directory into their main repository. In other words, we can say, they need to have repository of a repository. Git provide one method to handle such situations which is called submodule. Using submodule command in Git, we can define some sub-directories as a dedicated repositories. To declare that, lets assume that we have a repository of some C source codes and inside this C source code repository we have a sub-directory that contains some documentation and guides about the C source codes. The situation now is that we need to define this documentation sub-directory as a dedicated repository as it will undergo to many versioning. The case now is we have repository of C source code which contains another repository of some documentation.

12.1 Adding Submodule

We can define a sub-directory as a submodule of the main repository. This can be done using the following command

```
$git submodule add <URL> <local_path>
```

where URL is the remote path form where we can clone the sub-repository, local_path is the path in which the sub-repository resides in our local machine. The URL is optional if we know that we work locally only. After that, you can add and commit changes into your main repository as usual.

12.2 Clone Submodule

In case that you need to clone a repository that already has submodule, there are two way, either to simply use the following command

```
$git clone --recurse-submodules <path_of_main_repository>
```

or you can do it in two steps

```
$git clone <path_of_main_repository>  
$git submodule update --init --recursive
```

or even it can be done in three steps

```
$git clone <path_of_main_repository>  
$git submodule init  
$git submodule update --recursive
```

note that `--recursive` is optional, it is used if you know that your submodule has other submodules.

12.3 Get Submodule Updates

Since the submodule is another repository, it can happen that this submodule repository got changes in the remote source but the local version didn't get these changes. We need then to update the submodule with these remote changes. This can be done using

```
$git submodule update --remote <submodule_repo>
```

Or an easy way is to deal with the submodule as a normal repository, so navigate to it and pull the updates

```
$cd <submodule_repo>  
$git pull
```

12.4 Push Submodule Updates

If the case is that we have local changes in the submodule repository and we need to push it to remote repository, then we need to commit the changes first in the submodule repository and push it

```
$cd <submodule_repo>  
$git commit -am msg  
$git push
```

or if you want to push it only when you push changes of the main repository, you use the following

```
$cd <submodule_repo>  
$git commit -am msg  
$cd -  
$git commit -am msg  
$git push --recurse-submodules=on-demand
```

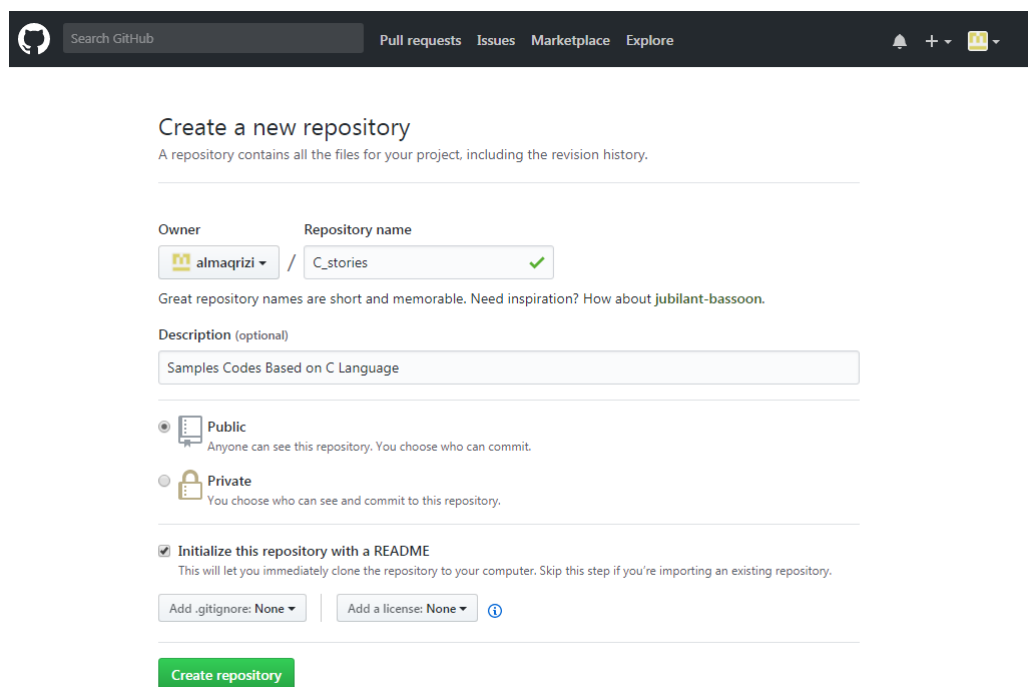
13 GitHub

13.1 Basic Operations

GitHub is one of the important tools that are built to provide Git as an interface for users. It became wide platform for publishing source codes for millions of users as it is a web-based interface. Users can clone easily any public repository using http protocol. It can be accessed at this link

<http://github.com>

You can create account simply and start building repositories. Main operations that were discussed before using command line interface can be operated in this web interface also. Figures 6, 7, 8 show graphical interfaces for creating repository, adding file and commit.



The screenshot displays the GitHub 'Create a new repository' interface. At the top, there's a navigation bar with the GitHub logo, a search bar, and links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. The main heading is 'Create a new repository', followed by a subtext: 'A repository contains all the files for your project, including the revision history.' Below this, the 'Owner' is set to 'almaqizi' and the 'Repository name' is 'C_stories', which is marked as valid with a green checkmark. A tip suggests repository names should be short and memorable. The 'Description (optional)' field contains 'Samples Codes Based on C Language'. Under the 'Visibility' section, 'Public' is selected, with a note that anyone can see the repository. The 'Private' option is also available. A checkbox for 'Initialize this repository with a README' is checked, with a note that it will clone the repository to the computer. At the bottom, there are dropdowns for 'Add .gitignore: None' and 'Add a license: None', and a green 'Create repository' button.

Figure 6: Create Repository in GitHub

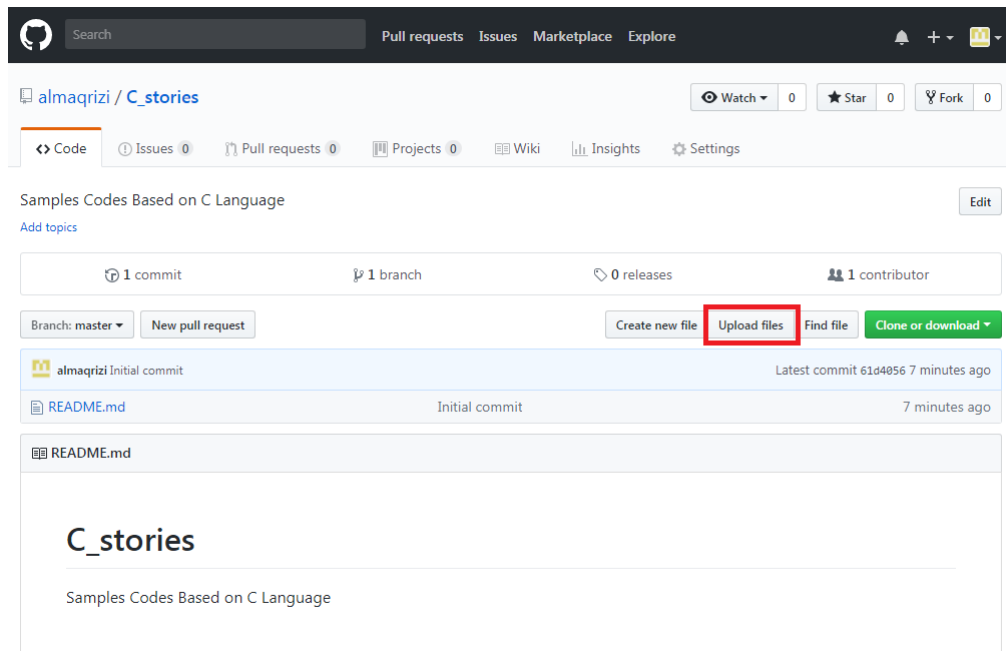


Figure 7: Adding File in GitHub

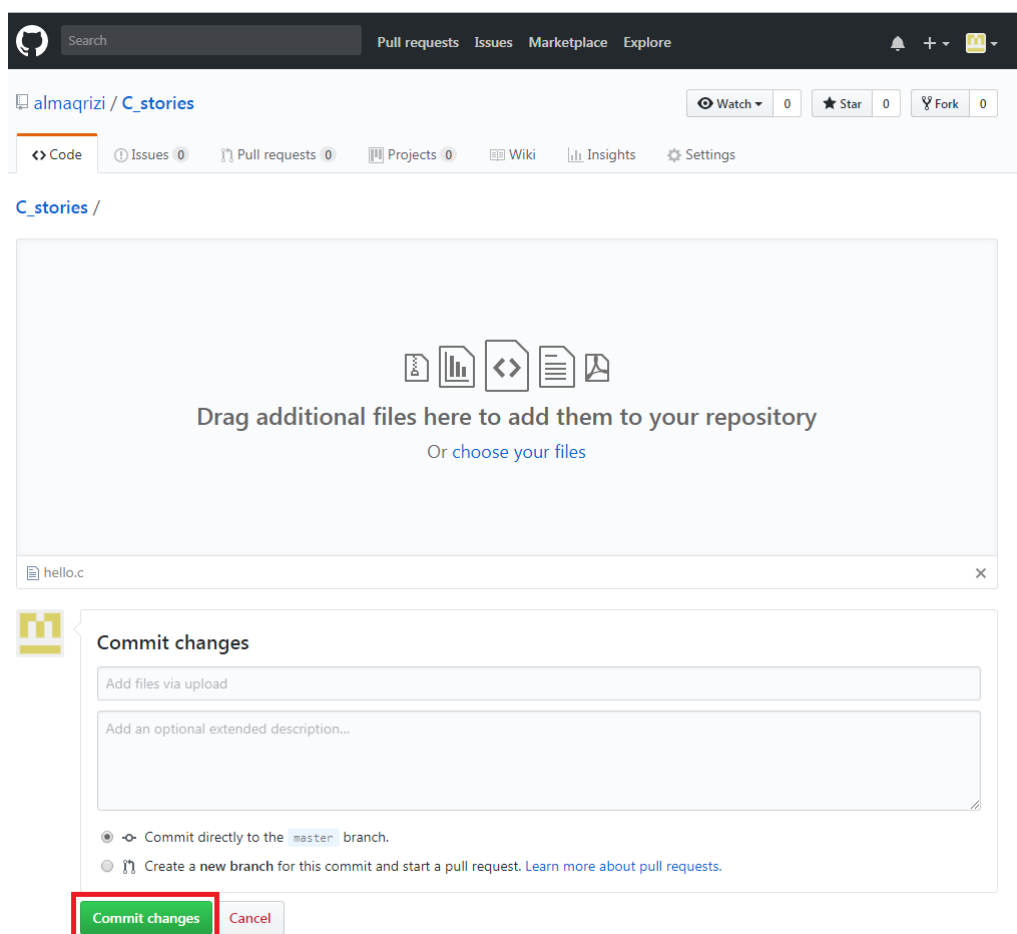


Figure 8: Commit File in GitHub

13.2 Fork and Pull Request

Created repositories in GitHub can be created to be private (no one can see or access the repository) or created as public (any person can clone the repository). Lets assume now for the case of a public repository, a ForeignUser wants to clone certain repository from OwnerUser account in GitHub.

As long as the ForeignUser can see the repository (public repository), he can download or clone repository from GitHub. But in this case it is not possible to push some changes back to the origin repository. To do this action in GitHub, ForeignUser has to use the fork option. ForeignUser has to register an account first in GitHub. After that he can search for the target repository to fork it. For example you can search for repository called “TCL_Stories” from a user called “electgon”, then you can fork this repository into your space. Figure 9 shows how to fork a repository in Github

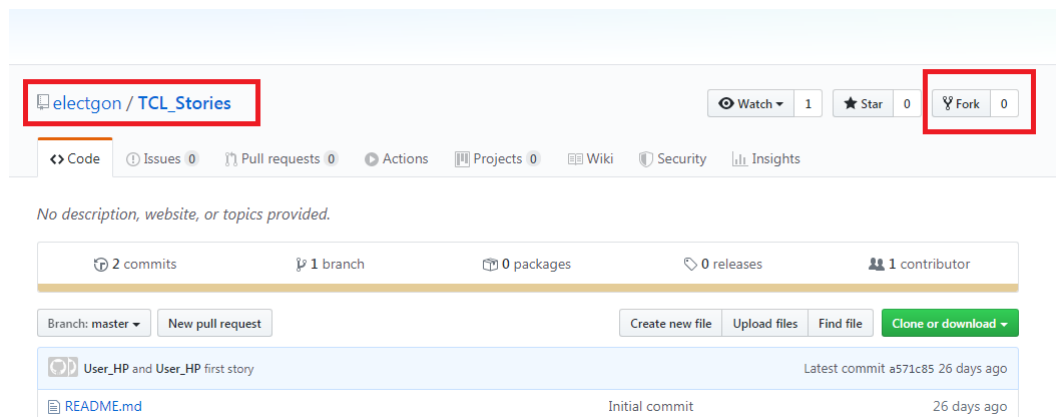


Figure 9: Fork a Repository in Github

When you click on 'fork' you can notice now that you have mirror of this repository in your space.

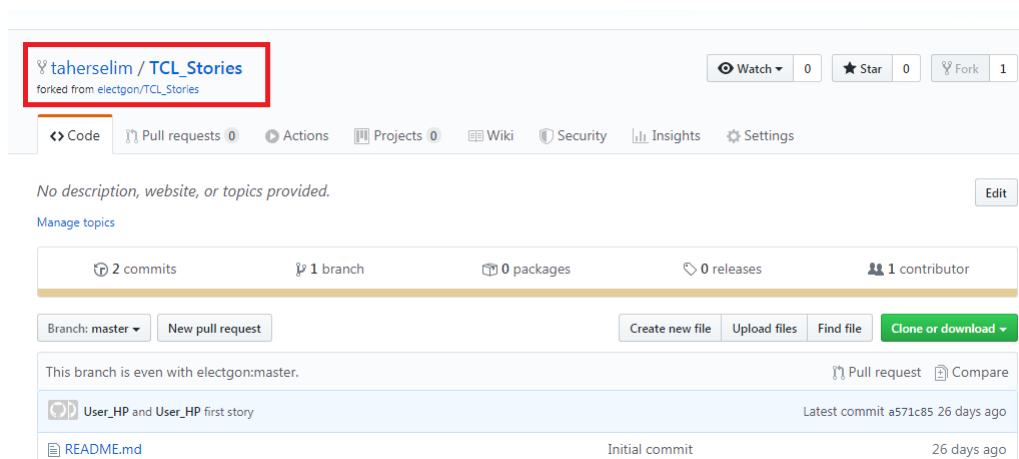


Figure 10: Forked Repository

Then you can clone or copy this repository into your machine

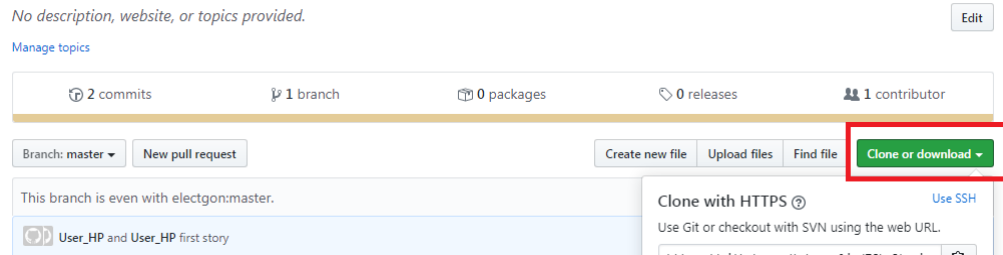


Figure 11: Clone Forked Repository

Afterwards, you can make your needed modifications in this cloned repository and add, commit, push the changes back to your forked repository. Now you can create **Pull Request**. Which means, you ask the original owner of the repository ("electgon" in our example) to merge your changes to his changes.

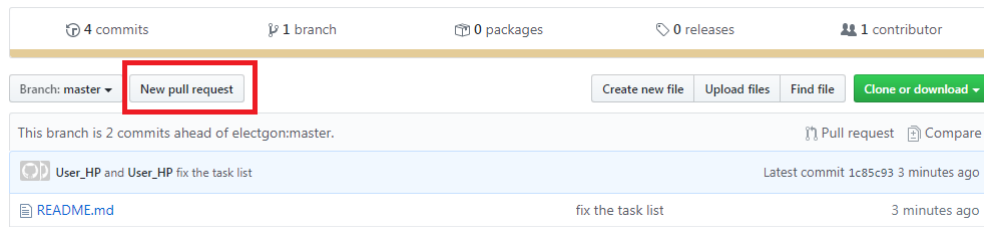


Figure 12: Create Pull Request

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

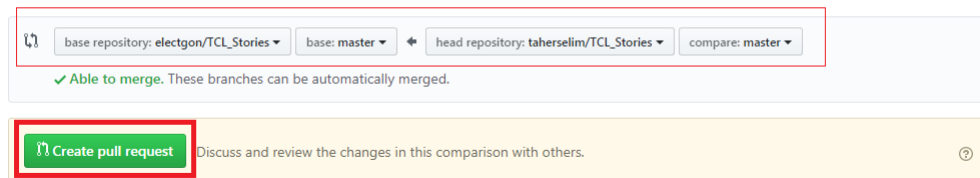


Figure 13: Start Pull Request

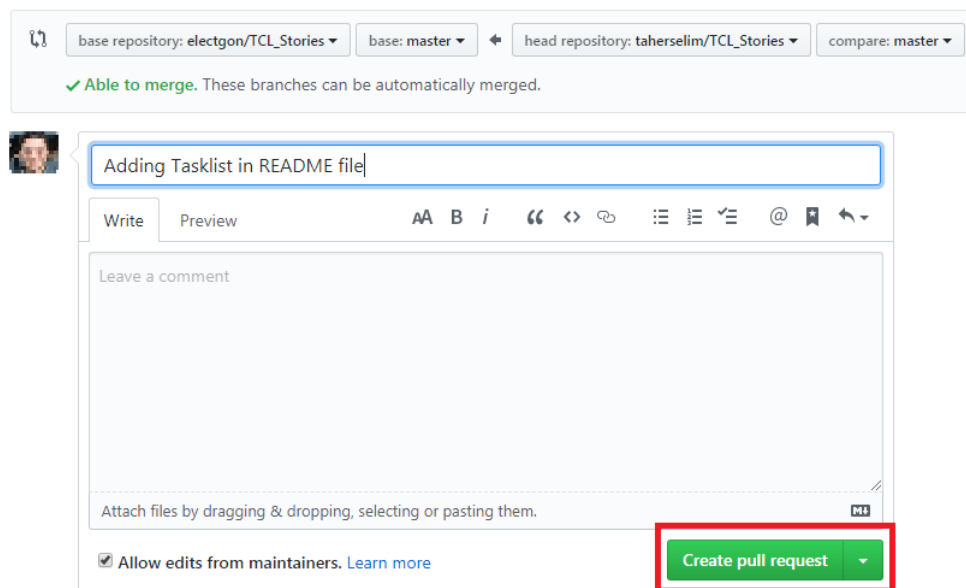


Figure 14: Write Comment for Pull Request

It is not enough to click on the Pull Request only, Actually you have to notify the original owner that you wish him review your changes, this has to be done from the right side as follows

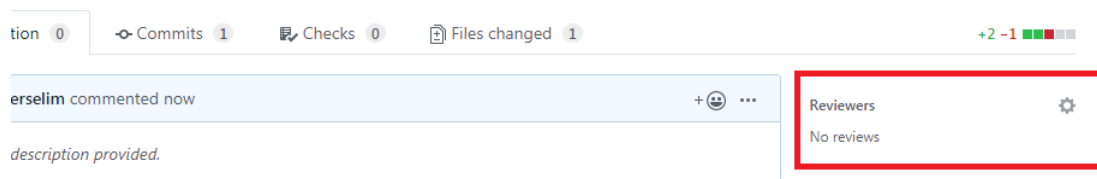


Figure 15: Add the Owner as Reviewer

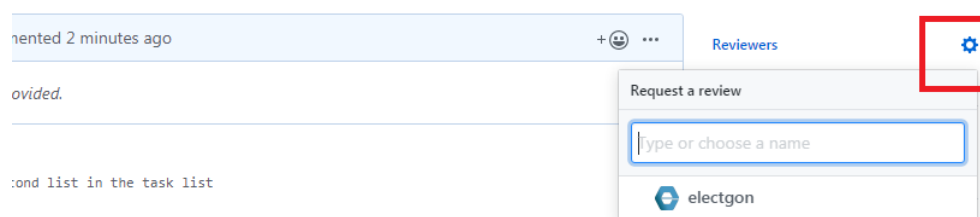


Figure 16: Send Review Request

The owner of origin repository then will get a notification that a change has been made to the repository and he is requested to make pull for the repository in order to accept the changes. He can accept the changes by merging it with the master branch or rejecting it by simply ignoring this branch. When the owner accepts your changes, you will be notified

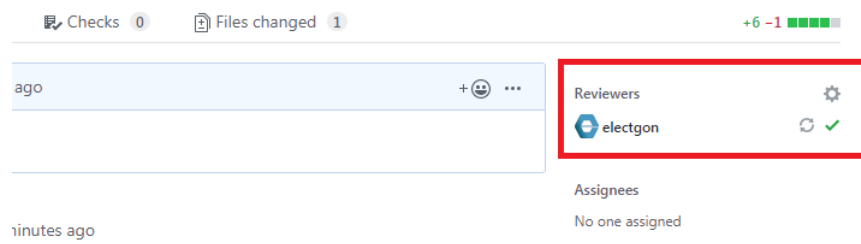


Figure 17: Confirmation of Changes Acceptance

14 Use Cases

In this section we will try to demonstrate some use cases that could be used by some developers to shed more light on how work flow using Git can be done.

14.1 Create Repository at Server from Client

This is basic scenario that can be used to initialize a repository. It assumes here that user wants to have central repository at a server machine from which different users can clone it. The idea may be in initialization of the server repository. If the design files reside at one of the client machines, no need to transfer these files to the server in order to create the repository. It is enough to create a bare repository at the server using the following command

```
$cd path/to/server/repository
$git init --bare RepoA
```

Now to update this repository from client machine, we can clone it to the client machine or create normal repository at the client machine, add needed files, push the changes to the server repository

```
cd myproject
$ git init RepoA
$ cd RepoA
### Import Repository files to add it
$ git add .
$ git commit -m 'initial commit'
$ git tag -a v1.0 -m 'starting version'
$ git remote add origin /path/to/ServerMachine/
$ git push origin master --tag
```

In this way repository at server machine will be updated. The following script can be used to automate the process from a shell terminal

```
#!/bin/bash
Repo_Path="./Test/"
Repo_Name="RepoB"
RemoteRepoName="RepoC"

RemoteUserName="serverAccount"
RemoteHostName="192.168.178.87"
RemoteOperationPath="C:/Personal/git_tut/workspace_04"
RemoteOperation="cd $RemoteOperationPath;
                pwd;
                git init --bare $RemoteRepoName"

ssh -o StrictHostKeyChecking=no ${RemoteUserName}@${RemoteHostName} ${
    RemoteOperation}

echo "Repo Name will be $Repo_Name"
git init $Repo_Path$Repo_Name
cd $Repo_Path$Repo_Name
```

```
cat > hello.c << "EOF"
#include <stdio.h>
int main(void){
    printf("Hello World \n");
    return 0;
}
EOF

git add hello.c
git commit -m "initial commit"
git tag -a v0.0 -m "starting version"
git remote add origin ${RemoteUserName}@${RemoteHostName}:${RemoteOperationPath}
}/${RemoteRepoName}
git push origin master --tag
```

14.2 Files Shared between Different Branches

Assume we have one simple SW application built using Python. This application is reading some database from xls file row by row. In each row the application is waiting for certain response from the user. Since this application is interacting with the user i.e. asking and waiting response from the user, there is one way to build this application to run in Python console. Therefore, we can add design files for this application in a Git repository. Assume then the following steps were followed to create the repository and add files

```
#!/bin/bash -e

## Prepare Repositories directories
BareRepo_Path="./BarePyRepo/"
Repo_Path="./ExamineUserApplication/"
Repo_Name="D_Py_Project"

mkdir -p $BareRepo_Path
mkdir -p $Repo_Path

## Create Bare Repository
git init --bare $BareRepo_Path$Repo_Name

## Create Local Repository for First Developer
cd $Repo_Path
git clone . $BareRepo_Path$Repo_Name
cd $Repo_Name

cat > hello.py << "EOF"

print('Hello World')

EOF

## Initialize Repository with simple hello file
git add hello.py
git commit -m "initial commit"
```

```
git tag -a v0.0 -m "starting version"
git push origin master --tag

## create woring branch for First Developer
git checkout -b cmd_mode

## Commit Design files of First Developer
cp ../../*.py .
cp ../../*.xlsx .

git add *.xlsx
git add Revise_* git
add ReviseCMDmode.py
git commit -m "adding design files"
```

Previous procedure has added and committed some Python files. These files can be listed by

```
$git ls-files
NewWords.xlsx
OldWords.xlsx
ReviseCMDmode.py
ReviseWords.xlsx
Revise_ask.py
Revise_init.py
Revise_parse.py
Revise_result.py
hello.py
```

So it consists mainly of some xlsx files which contain our database, and some Python files which perform needed functionality of this application. The main execution of these Python files resides in file ReviseCMDmode.py. This ReviseCMDmode.py is calling other functions from the other associated Python module.

Assume now that there is a request to have GUI for this application and another developer has worked on this request. The GUI module results then in the following design files.

```
ReviseGUImode.py - Revise_ask.py - Revise_init.py - Revise_parse.py -
Revise_result.py
```

In which the developer has used previous modules of the CMD mode but replaced the ReviseCMDmode.py with the new GUI mode file ReviseGUImode.py. Now assume GUI developer has used the following script to clone the repository and add his files.

```
#!/bin/bash -e

## Prepare Repositories directories
BareRepo_Path="./BarePyRepo/"
Repo_Path="./GUIUserApplication/"
Repo_Name="D_Py_Project"

mkdir -p $Repo_Path
```

```
## Create Local Repository for Second Developer
cd $Repo_Path
git clone . $BareRepo_Path$Repo_Name
cd $Repo_Name

## create working branch for Second Developer
git checkout -b gui_mode

## Commit Design files of Second Developer

git add ReviseGUImode.py
git commit -m "adding design files"
```

CMD developer has now master branch and cmd_mode branch. While GUI developer has master branch and gui_mode branch. Assume now that both have pushed their changes to the origin repository.

```
## CMD Developer
$git push origin cmd_mode
.....
## GUI Developer
$git push origin gui_mode
```

To let each of them to get the other branch, they have to fetch it. For example if the CMD developer issued

```
gitk --all
```

He will get the following graph

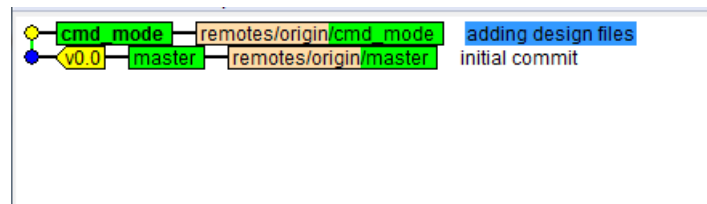


Figure 18: CMD Repository Before Branch Update

In order to update his repository, he has to issue the following commands

```
$git ls-remote --heads ## This step to list remote branches
$git fetch origin
$git checkout --track origin/gui_mode
```

Displaying new updates now using gitk will result in the following

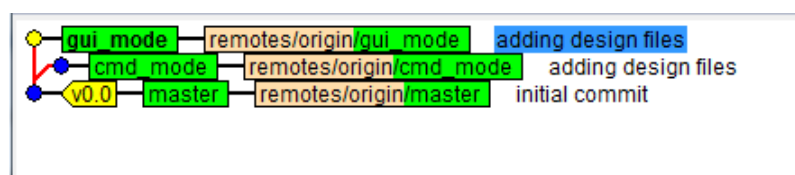


Figure 19: CMD Repository After Branch Update

The last step has to be done also for the GUI developer if he wants to get cmd_mode branch. GUI developers needs the following design files which is originally located at cmd_mode branch

```
Revise_ask.py - Revise_init.py - Revise_parse.py - Revise_result.py
```

GUI developer can get it by simply using the following command

```
$git checkout cmd_mode Revise_ask.py Revise_init.py Revise_parse.py  
Revise_result.py
```

In that command, GUI developer has used same design files that reside in cmd_mode branch. If any changes took place in these files, GUI developer has to re-issue previous commands again in order to get the updates.

14.3 Check Status of Specific File

In last step of previous example, we needed to update gui_mode branch with design files from cmd_mode branch. To check whether there are changes in these file at the cmd_mode branch we can use the following command

```
$git diff --name-only cmd_mode gui_mode *.py
```

The displayed files will be marked with one of the following letters if there are any change
D: which means that file exist in cmd_mode branch (as it is written firstly).

M: if the file is modified.

A: if the file exist in gui_mode only.

14.4 Branch Merge Conflict

It can happen that we need to merge two branches to consolidate design files. It can happen also that two branches have the same file but with different versions. In this case, Git will issue a conflict error and pause the merge operation until user resolve the conflicting part. To discuss this, assume our previous example in which we have branches master, cmd_mode and gui_mode for a Python based application. This application has design files for GUI operation contained in gui_mode branch. It has also design files for command line operation contained in cmd_mode. Lets assume now that we need to combine both branches (cmd_mode & gui_mode) into the main branch (master). First we merge gui_mode branch into master branch as follows

```
$git checkout master  
$git merge gui_mode
```

At this step, for demonstration purpose, some modification has been made to file Revise_init.py of the gui_mode branch. The same file is contained also in cmd_mode but not updated. Then we need to merge content of cmd_mode branch to the master branch. We should get error as below

```
$git merge cmd_mode  
Auto-merging Revise_init.py  
CONFLICT (add/add): Merge conflict in Revise_init.py
```



```
Automatic merge failed; fix conflicts and then commit the result.
```

By opening the file which has conflict, we can find the conflicting part is labeled in the file as below for example

```
<<<<<< HEAD
    Table_Length = master_sheet.max_row
=====
    while master_sheet.cell(row=k, column=1).value!=None:
        k=k+1
        Table_Length=k
>>>>>> cmd_mode
```

Which indicates that developer has replaced part of the code. The corresponding content of cmd_mode branch is indicated between (=====
>>>>>> cmd_mode).

At this step user can break the merge operation by typing the following command

```
$git merge --abort
```

Then he can update the cmd_mode branch with the changes then merge again

```
$git checkout cmd_mode
$git checkout gui_mode Revise_init.py
$git checkout master
$git merge cmd_mode
```

Or after breaking the merge, he can revert the first merge operation between master and gui_mode using the following command.

```
$git revert HEAD
```

Or without breaking the merge operation, user can resolve the labeled part in file manually then add the file again and commit

```
$git add Revise_init.py
$git commit -m "fixing conflict"
```

14.5 Merg Branches in GitHub

In case that we need to move our local repository to GitHub this can be done in simple steps. First we need to create space for the local repository in GitHub. Then we push the local repository to the created repository in GitHub. Lets continue with last example to see how this can be done.

First step is to create repository in GitHub. Note that you have to avoid committing initial file to the create repository so that when you push the local repository, you will have exact copy of the local repository in GitHub.

Then in your local repository add the link of the created GitHub repository as a remote repository

```
$git remote add GitHubProj https://github.com/UserAccount/RepositoryName.git
```

Note that *UserAccount* denotes here the name of your GitHub account. So this shall be filled correctly. Also *RepositoryName* should be your GitHub repository name.

Now you can push the local repository to GitHub

```
$git push GitHubProj --all
```

After issuing last command you will be asked to provide *UserAccount* name and password. After providing it successfully, your project shall be hosted now in GitHub.

For contribution between several developers in GitHub, Fork and Pull Requests are used as discussed before.

Then in order to make changes in your GitHub repository, it is advised to detach new branch to contain your changes. This can be done in GitHub in the Branch drop-down list as shown in figure 20

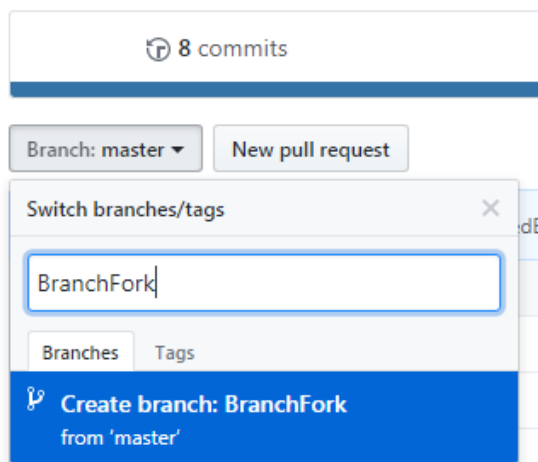


Figure 20: Create New Branch in GitHub

Then in your target file, you can open it and edit it.

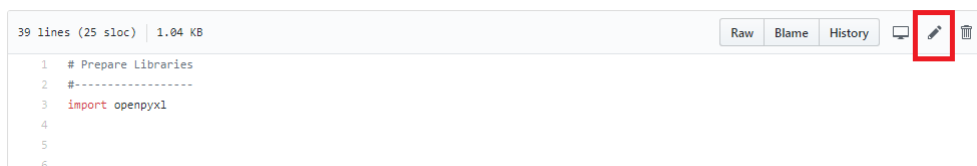


Figure 21: Edit File in GitHub

After that, you can commit your changes.

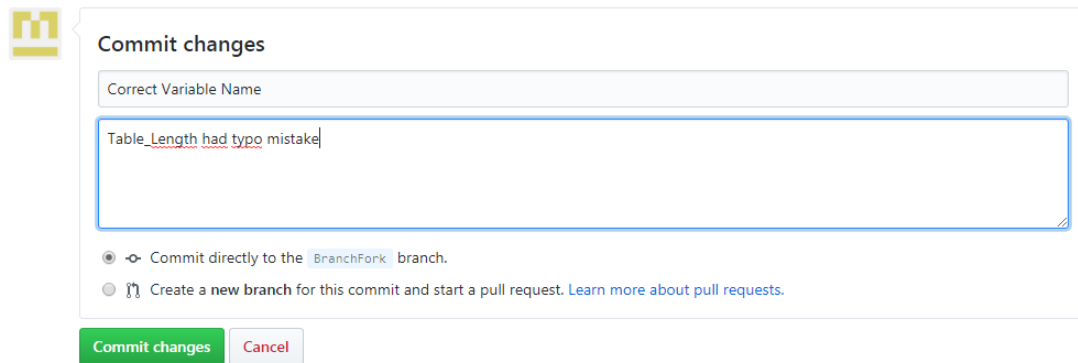


Figure 22: GitHub Commit Changes

GitHub will detect that you have made changes in the repository and will prompt you if you need to create Pull Request for these changes as shown in figure . You can create Pull Request then.

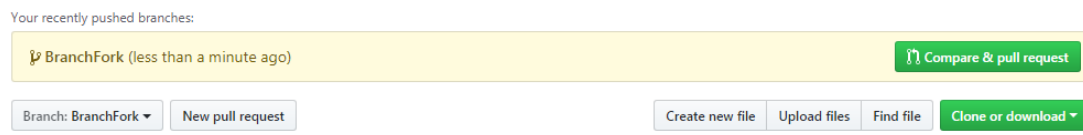


Figure 23: Pull Request Button

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

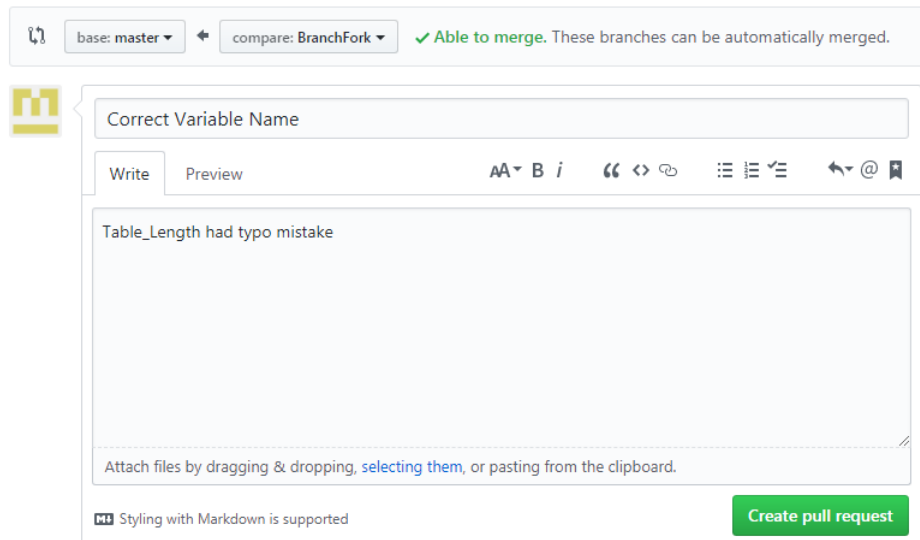


Figure 24: Create Pull Request

Owner of the Repository will be notified then with this Pull Request in Pull Request tab.

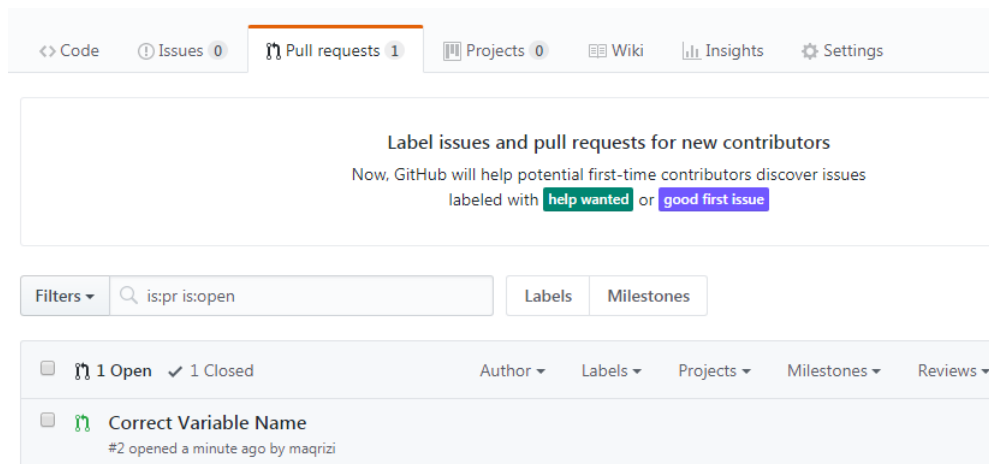


Figure 25: Pull Request Notification

He has the option now to accept the request by merging it to the main branch or discuss further with the requester or ignore this request.

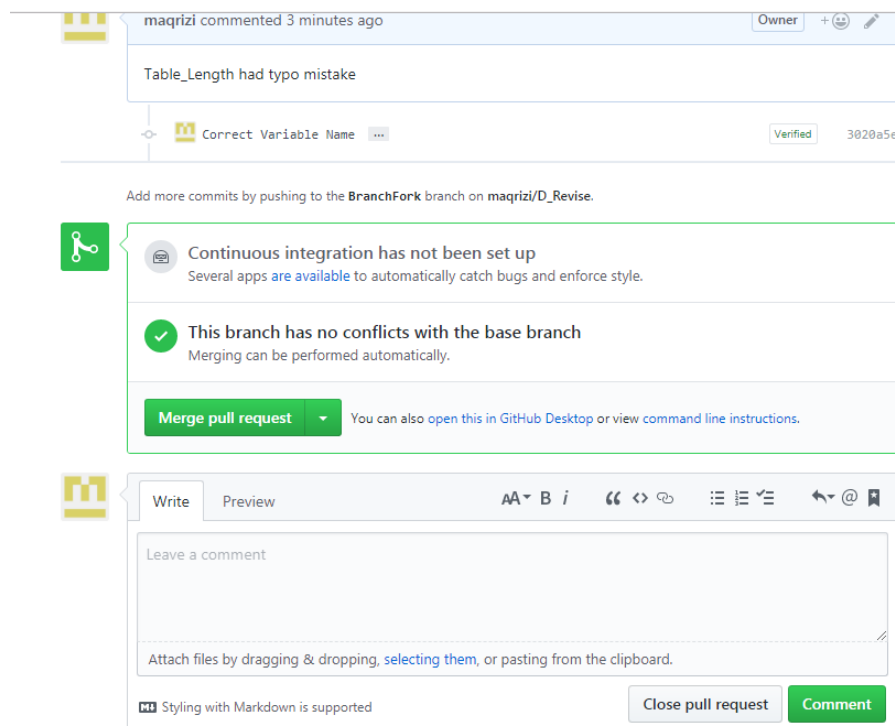


Figure 26: Pull Request Review

15 Summary

Git as a versions management tool provides various options and facilities to ease versions management tasks during development of projects files. It is the turn of the user to get enough experience and learn how to use this tool. In table 1 we can check samples of git commands that have been used during this tutorial to summarize important operations used in Git.

Git Operations Samples

	Command Sample	Notes
Configure Repository	\$git config --global user.name "Working Tester" \$git config --global user.email tester@test.tst	*Without global option, you can configure for local repo.
Review Configuration	\$git config user.name \$git config --list	-
Create Repository	\$git init <name_of_created_repository> \$git init --bare <name_of_created_repository>	*Use --shared option if many users are contributing to this repository
Clone Repository	\$git clone <RepoPath> <RepoName> \$git clone u_name@<ip.add.re.ss>:<RepoPath>	
Repository Status	\$git status	-
Adding/Committing	\$git add <filename> \$git commit -m "message" \$git commit --amend	*Use --all with add to add all files. *Use --amend with commit to commit to last commit (don't set new commit)
List Files	\$git ls-files	-
Remove Files	\$git reset filename \$git rm --cached file_name	*reset to remove stage file *rm --cached to remove committed file
Remove File From Old Commits	git filter-branch --force --index-filter "git rm \ --cached --ignore-unmatch path/to/file" \ --prune-empty --tag-name-filter cat -- --all	* use "git push origin --force --all" to update remote repository with this removal.
Ignore Files	\$cat > .gitignore << 'EOF' *.o EOF \$git commit -am "adding ignore file"	*"#" can be used to write comments in the ignore file.
List Remote Repositories	\$git remote -v	-
Add Remote Repositories	\$git remote add <RepoName> <RepoPath>	-
Push/Pull/Fetch	\$git push <RemoteName> <BranchName> \$git pull <RemoteName> <BranchName> \$git fetch <RemoteName> <BranchName>	*Use --all option if you want to push all branches
Review Local Branches	\$git branch	-
Review Remote Branches	\$git branch -r	-
Review Branch Content	\$git ls-tree -r --name-only branch_name	-
Create Branch	\$git branch branch_name	-
Switch Branch	\$git checkout branch_name	-
Delete Branch	\$git branch -d branch_name	-
Merge Branch	\$git merge branch_name	*Use --abort option to break merging
Merge Commit	\$git cherry-pick commit_number	
Track Branch	\$git checkout --track origin/TrackedBranch	use "git branch -vv" to see all tracking
Tagging	\$git tag -a tagName -m "message"	-
Switching	\$git checkout hashNum \$git checkout branchName \$git checkout TagName \$git revert HEAD	*Use \$git checkout - to go back where you were before you checkout
View Logs	\$git log --pretty=format:"%h: %s" -3 --name-only \$git log --graph --oneline --decorate --all	-
View Difference	\$git diff filename \$git diff --name-only branch1 branch2 filename \$git difftool --no-prompt --tool=meld hello.c \$git difftool --dir-diff --tool=meld hash_1 hash_2	-
Add Submodule	\$git submodule add <url> <local_path>	-
Clone Submodule	\$git clone --recurse-submodules <main_repos> or: \$git clone <path_of_main_repository> \$git submodule update --init --recursive	-
Push with Submodule Updates	\$git push --recurse-submodules=on-demand	-

Table 1: Git Commands Sample

Bibliography

[1] Source: "<http://www.techterms.com/definition/repository>".

[2] Source: "http://www.techterms.com/definition/version_control".