Git Manual

Contents

..... 1

General Information 2

Creating a Repository 2

Workflow 4

Workflow Commands 4

Branches 5

Branch Commands 6

Conflicts 8

Mistakes 9

General Information

Git is a Version Control System (VCS) which takes a snapshot of all files every time they are committed and stores every version of those files so that you can look back over every step of the work done within a repository. Every version of every file is also stored on each computer with access to the repository. Since each computer is working on their own copy of the version history, and they can modify their copy as much as they want without worrying about affecting the repository itself.
With the "pull" command you update your copy to the newest one in the repository, while "push" modifies the repository with your changes.

Files in git can have three *stages*:

1. Modified (The file has been changed since it was last committed)
2. Staged (The file is chosen to be included in the next commit)
3. Committed (The file is now stored in your local copy)

There are different ways to authenticate yourself with git, either you use HTTPS in which you must write in your username and password every time, or you set up an SSH key on your computer which git checks every time you commit saving you the input.

Creating a Repository

git init (This adds a new git repository at the terminals current location)

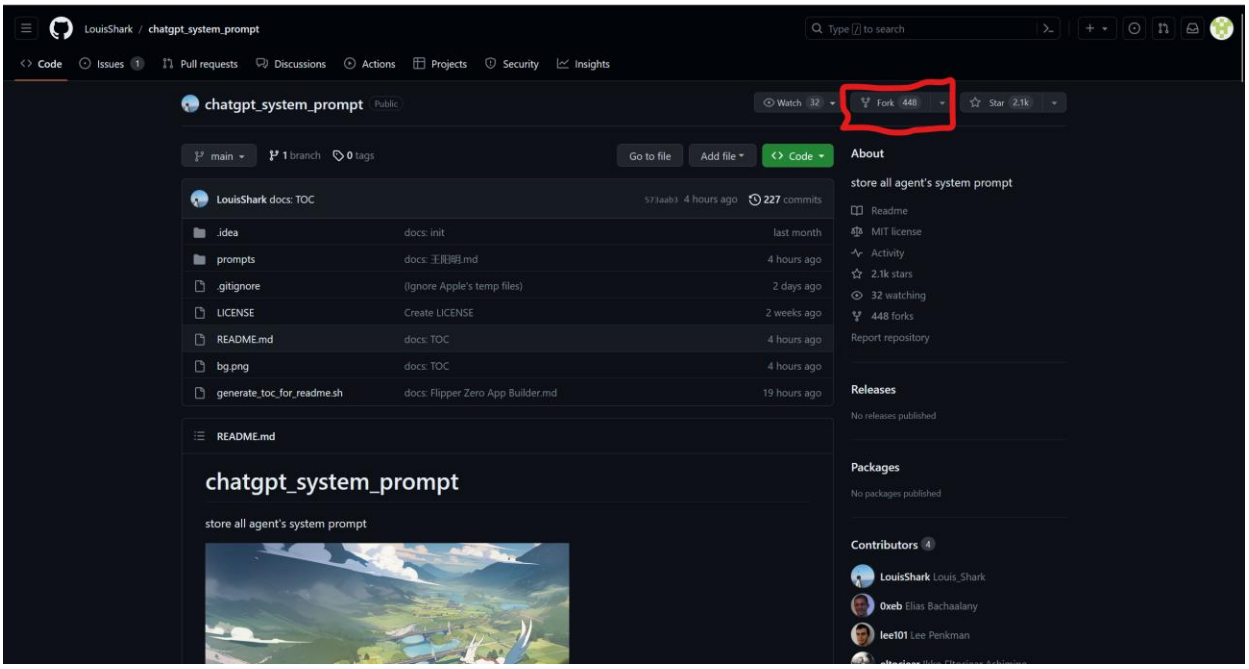
git clone */path/to/repository* (This creates a working copy of an existing repository)

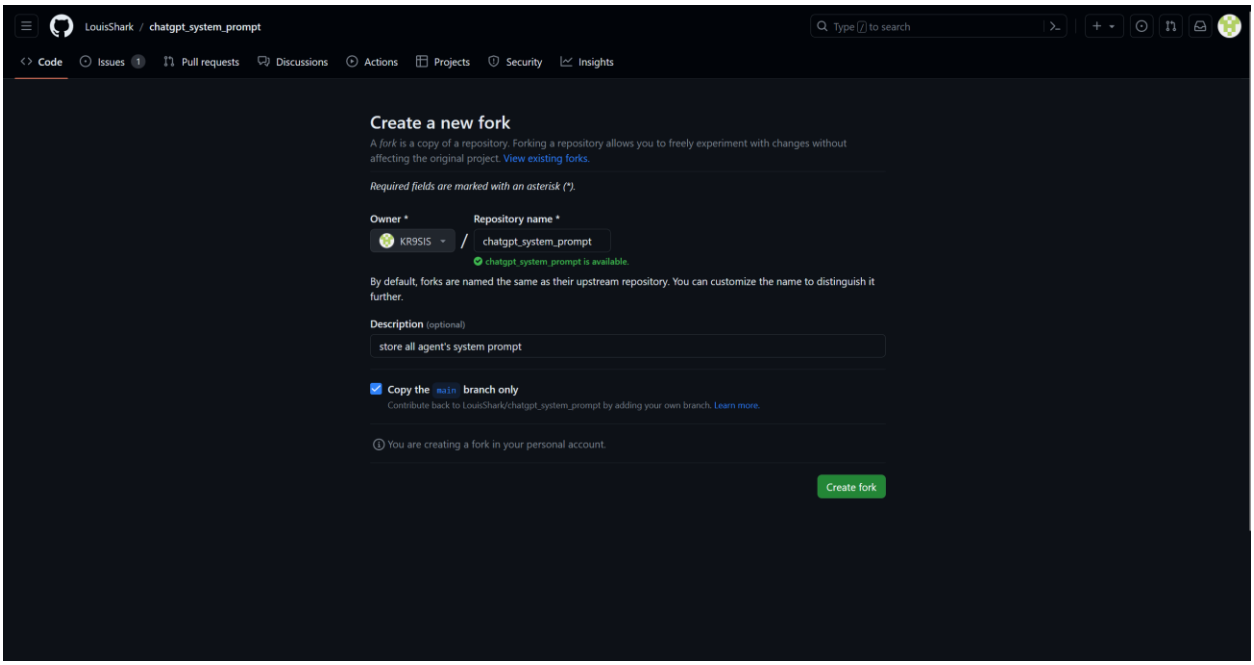
Although when using a remote server, the command will be:

git clone username@host: /path/to/repository

If on the other hand you want to copy someone else's project, then you fork it on github.com.

You do this by finding a open source repository and pressing the Fork button located here:





Whereafter you simply modify the repo as desired and create the fork by pressing the green button.

Workflow

Git has three "trees" containing your work.

1. *Working directory*. This is where all files are stored, this is also where you send files with "push".
2. *Index*. This is a staging area where files are temporarily stored with the "add" command.
3. *Head*. This points to the last "commit" which was made.

Workflow Commands

To see current working information, you use:

git status

This shows you:

1. What branch you are on.
2. If you are in synch with the remote repository,
3. What files are tracked or untracked.
4. IF tracked files, it tells you what *stage* they are in.

You can propose changes and add items to the Index with:

git add *filename* *(To stage specific file)*

git add *filename filename etc.* (To stage two or more files)

git add * (* means everything in the *working directory* is staged to the *Index*)

This is the first step in working with Git, adding your changes for staging. To send them further and commit these changes you use:

git commit -m "*One short commit sentence.*"

This sends the changes to the *HEAD*, but not to your remote repository (cloud) yet. To send them there you use:

git push (Sends everything committed to current remote repository)

IF you've designated specific remotes and branches then:

git push *<remote> <branch>* *(Sends everything to a specific branch of a remote)*

If you did not designate specific remote or branch therein then the default names are remote "origin" and branch "master" or "main".

If you want to update your local copy of the project from the remote repository, you can use:

git pull (Downloads latest changes from remote repository AND MERGES your work with it)

git fetch (Downloads latest changes from remote repository WITHOUT MERGING your work with it)

If you want to see what changes you've made recently, you can use:

git diff (This prints out UNstaged changes made since last commit)

git diff –staged (This prints out STAGED changes made since last commit)

git diff HEAD (This prints out ALL changes made since last commit)

In each case a line preceded with a + was added to the file and each line preceded with a – was deleted from the file. If default colors are enabled the additions will be green and deletions will be red.

tldr: git add <filename> | git commit -m "*commit msg*" | git push

Running these three commands will send your modifications to the remote repository. The | (pipe) also works in git bash.

Branches

Git gives developers the ability to differentiate between branches with the default branch being main or master.[1] The *HEAD* is a pointer pointing to your current branch. Working code on main branch which runs the program which users can access and must work while the work-in-progress code such as new features or functions are stored on other branches of our making to not ruin the normal program. This also keeps merge collisions to a minimum.

[1] Every default branch I've seen has been called main, but every documentation I've seen calls them master, so let's just assume they're both used. I however will refer to this branch as main because that is what my PC calls it.

Branch Commands

To create a new branch, you use:

git branch *<branchName>* (This creates a new branch for your project)

git checkout *<branchName>* (This moves you to the specified branch)

git checkout -b *<branchName>* (This creates a new branch for your project AND you go there)

A branch you create is private unless you explicitly share it. Sharing it is not necessary but if you want to work with other developers on a specific branch then you use:

git push *<remote> <branchName>* (--set-upstream **or** -u)

You don't use parenthesis simply -u or --set-upstream. Remote is usually called *origin.*

Now that you have more than one branch, if you want to see the difference between two branches, then while within one of them use:

git diff *<branchName>* (Checks the difference between the branch you're on and the one you specify)

To merge two branches together you need to be in the branch you would like and merge to such that if you had two branches newFeature and development. You finish work on the new feature in the newFeature branch, then to merge the complete program you go to development and use:

git merge *<branchName>* # branchName == newFeature in our example

This takes the feature from newFeature and merges it into the development branch.

When you have finished merging a branch to development, then you delete it to clean up after yourself. To do so you use:

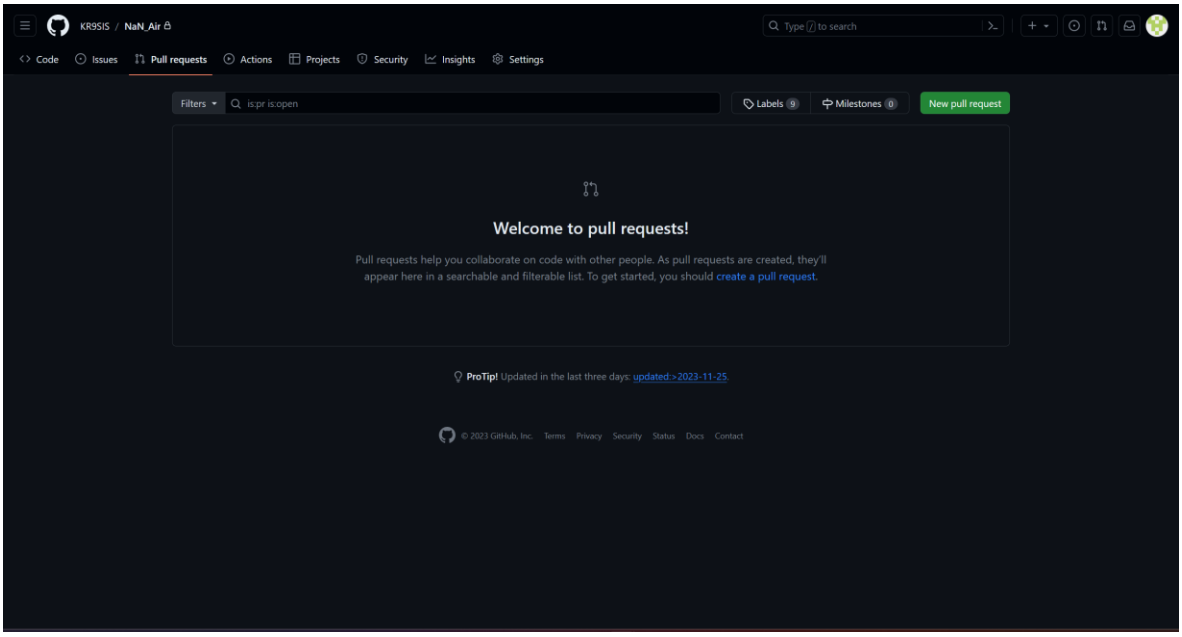
git branch -d *<name-of-branch>* (Deletes specified branch, also warns if branch isn't merged)

If that branch had been pushed to the remote repository as well, then you use:

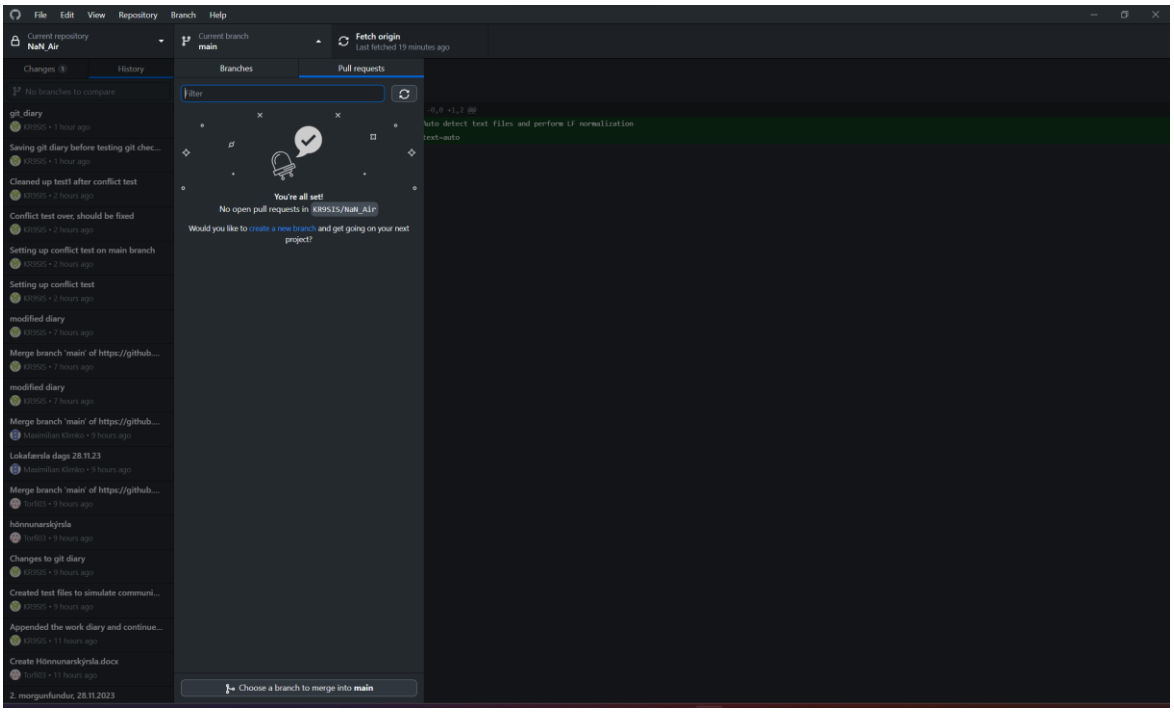
git push origin –delete *<branchName>* (This deletes the branch stored on the remote repository)

If you intend to merge features you've made into the main branch though, then you use Pull Requests which can be found on Github Desktop or github.com. These merges must be reviewed and approved by a different developer so as not to disrupt the system accessed by the users.

Pull request page on github.com:



Pull request tab on Github Desktop:



Conflicts

If two developers have in some way modified the same line in a file, then git will not know what to do and asks the developer trying to merge branches or push their modifications which to keep. A good example of this can be found in a YouTube video named: Verklegt námskeið II - Git - Git Intermediate.

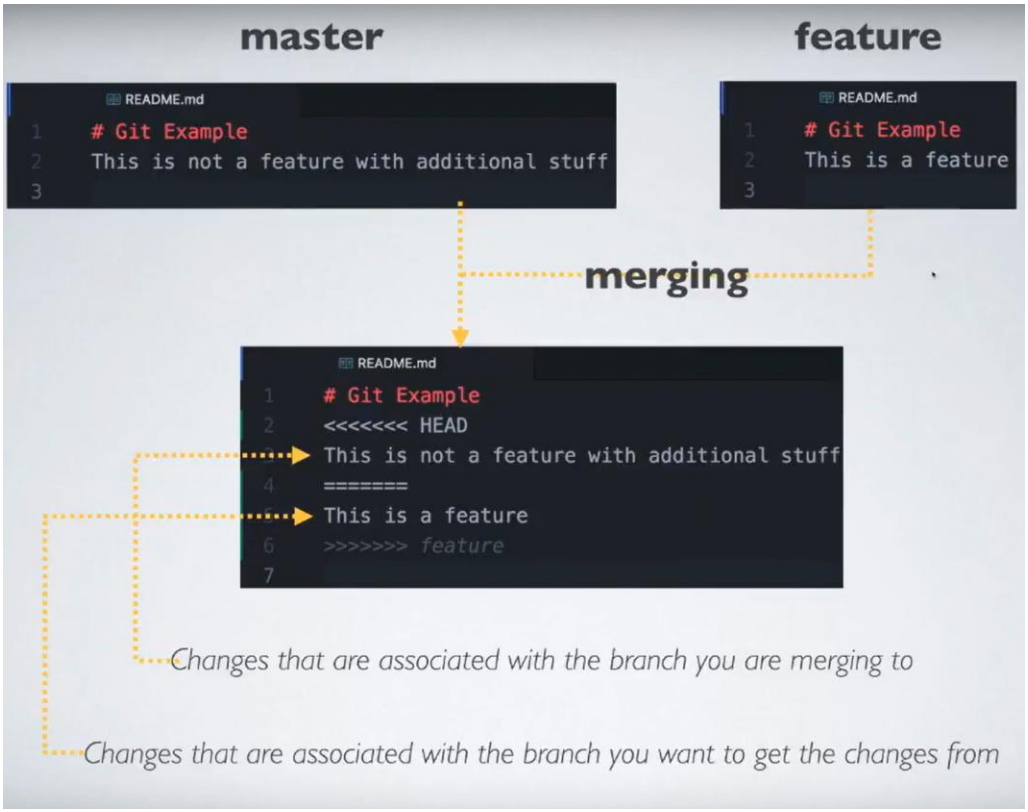


Figure 1 Screenshot taken from Arnar Leifsson's YouTube video at 17:29. Video link here.

What is happening here is that two different developers each in their own branch made two different modifications to line 2 of README.md. This means that when the branches are merged, a conflict happens, and git issues the following warning:

```
krist@KHSs-Yoga-7i MINGW64 ~/OneDrive/Code/Py/VN1/NaN_Air (main)
$ git merge conflictTest
Auto-merging src/logic_layer/test1.py
CONFLICT (content): Merge conflict in src/logic_layer/test1.py
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 2 Git warning from KHS PC

When we then look into test1 in the warning scenario it looks like the lower git example with the

<<<<< HEAD

“Branch1 content

=====

branch2 content

>>>>> feature.

So we must delete the conflict so that the only thing left is

“BranchX content”

before adding and committing the file again. When that is done, then we have resolved the conflict and can now delete the leftover branch.

Mistakes

Sometimes mistakes are made, and we stage, commit, or push information we didn’t want to.

git reset HEAD *<filename>* (This will unstage A FILE that has been staged)

git reset (This will unstage ALL changes and empty out the *Index*)

If you want to take back everything you’ve done since your last commit. Either for a specific file or entire directory, then you can use:

git checkout -- *<filename>* (Returns the file to how it was after the last commit made)

git checkout -- . (All files in directory will be set back to how they were after last commit made)

If you want to take it further and undo the last commit you made, then you can use:

git revert (New commit which will undo the changes made since a previous commit)

This does not clear the git history, rather it looks back through it, finds the previous commit and changes the current state of the project to the state it was in after previous commit, then commits those changes made to history.

git reset (This winds back time until a previous commit clearing the git history since)