# **What is Git**

Git is a distributed version control and source code management system

# **What is GitHub**

GitHub is a Git repository hosting service. It’s an online database that allows you to keep track of and share your Git version control projects outside of your local computer/server

# **Version Control System**

Version Control System (VCS) is a mechanism that helps software developers to collaborate and maintain a complete history of changes to their work.

By tracking and logging the changes you make to your code, a version-control system gives you the power to review or restore earlier versions as needed

Let’s say that you’re working on a web development project, and through the course of your revisions, you suddenly notice all of your text has become misaligned. So over here; instead of having to crawl back through every line of code, you can just use your version control system to reload earlier versions, until you pinpoint the offending change and correct it.

Following are the types of VCS −

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly publish their work. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository.

You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

# **Working Directory and Staging Area or Index**

The working directory is sort of like a workbench, it's where you work on your files (you edit them, you add new files, you delete files etc.).

The staging areas where Git starts tracking your files. Whenever you are done working on a file (or files) in your working directory, you copy them to the staging area (using the git add command).

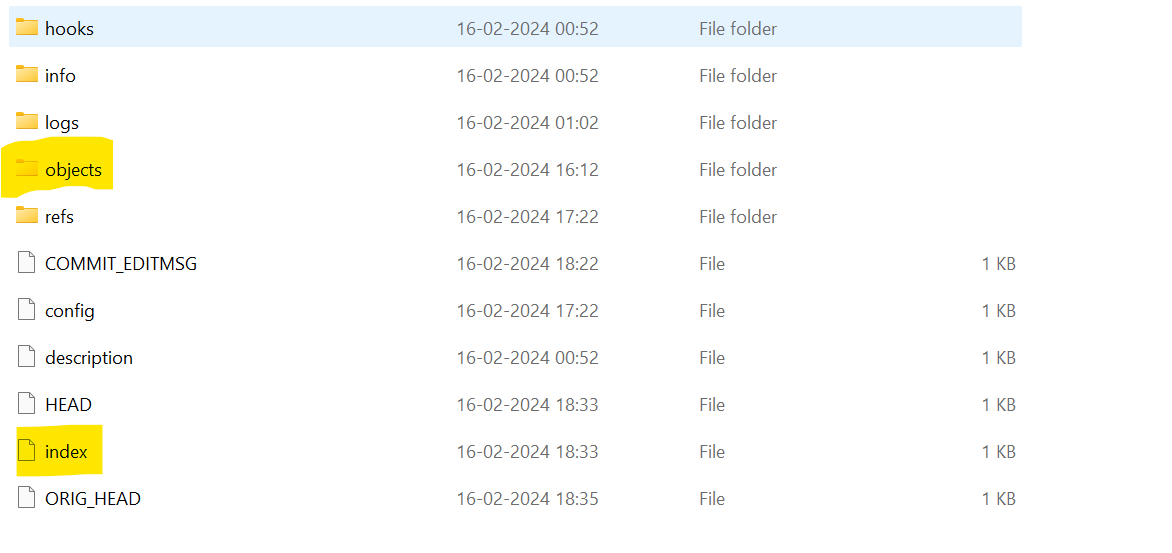
In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

# **Local Repository**

A local repository is a copy of the entire project's history and codebase that resides on a developer's machine.

The .git folder is what makes your repository a git repo. The .git folder contains all the information and the metadata that is necessary for your project in version control and all the information about commits, remote repository address, commit history etc.

Within the .git folder there are two "places" that should be mentioned, the staging area (represented by the index file) and the commit history (represented by the objects folder).



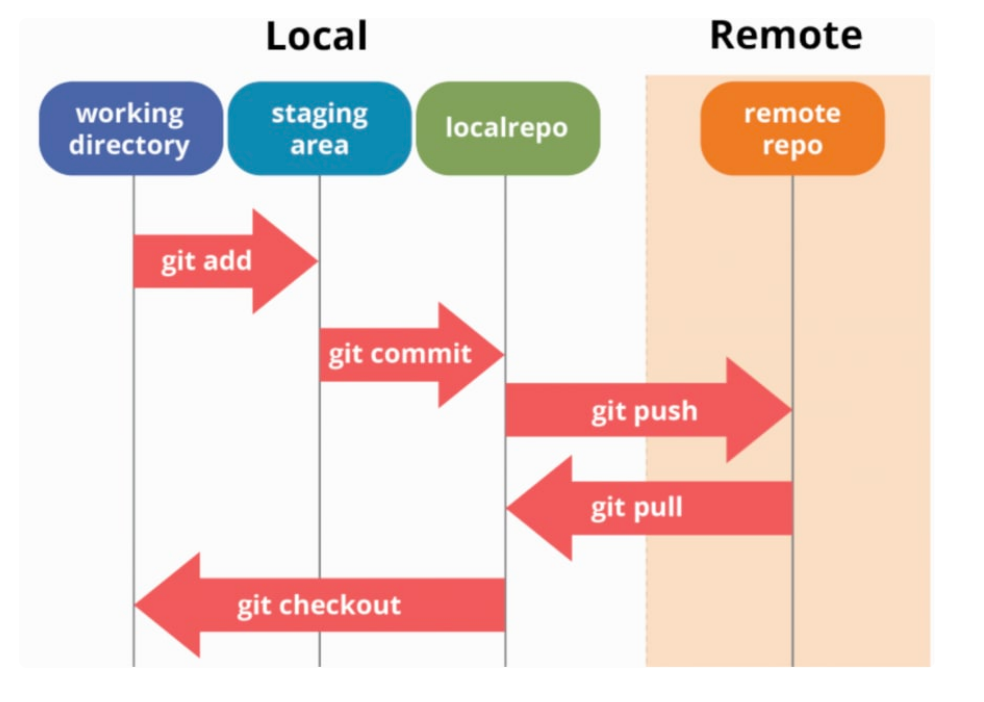
# **Basic workflow of Git**

Step 1 − You modify a file in the working directory.

Step 2 − You add these files to the staging area.

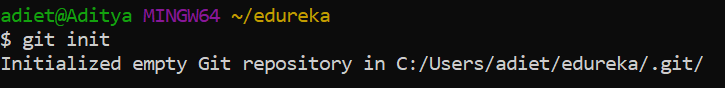
Step 3 − You perform commit operation that moves the files from the staging area.

Step 4 − After push operation, it stores the changes permanently to the Git repository.

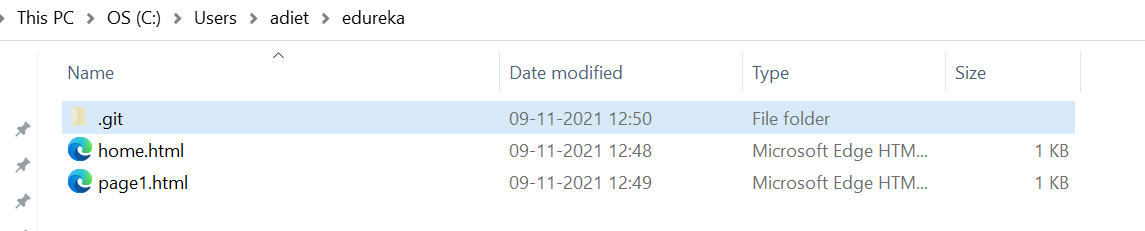


Step 1: Go inside the folder where you are creating your files and type **git init**

The git init command is the first command that you will run on Git. The git init command is used to create a new blank repository. Do remember that it by default creates it on master branch. To create it in another branch, use git init -b main (main here is the branch name)



After this, a hidden folder will be created in your folder. The git init command creates a .git subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata



Step 2: Add files to your staging area. type **git add**

The git add command is used to add file contents to the Index (Staging Area). Files will only be tracked once the file is added to staging area.

Every time we add or update any file in our project, it is required to forward updates to the staging area --- git add <file name>

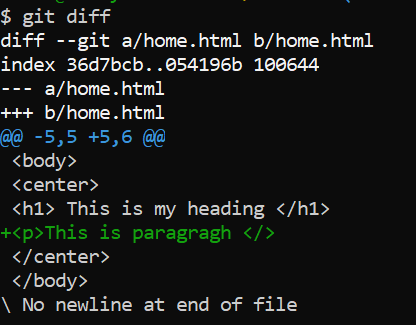


\*\*there shouldn’t be any spaces in file name otherwise git will return error.

To add multiple files use: **git add .**

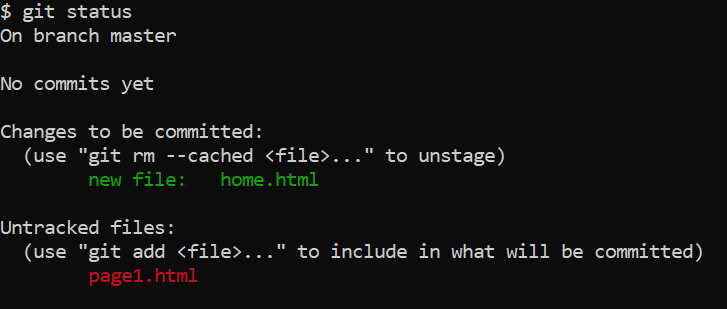
To undo an add operation, run the below command: **git reset <filename> or git reset .** to unstage all files

Lets say if you made few more changes after the file was loaded to staging area, you can check this by typing **git diff**



Use **git add** again to add the updated file to staging area.

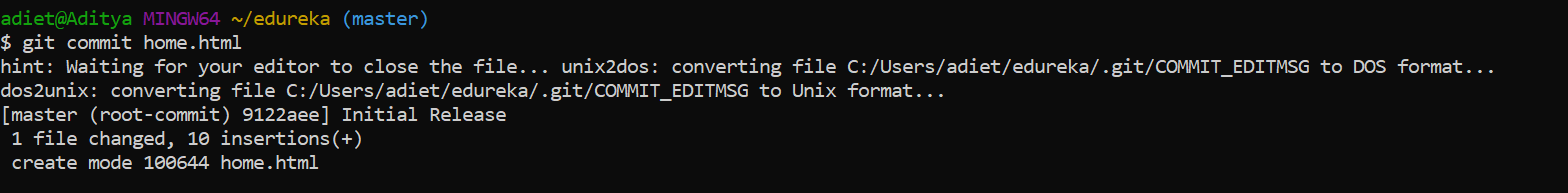
You can also use **git status** to know what all files have been committed, which files are currently present in staging area, which are absent.



Step 3: Now we need to commit the changes to a local repository. To do so use **git commit**

To commit selected files, use **git commit <file name>**

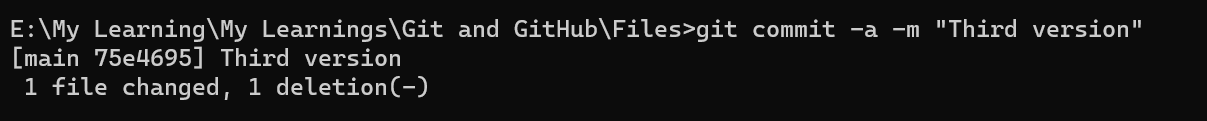
It is used to record the changes in the local repository. The staging and committing are co-related to each other. Staging allows us to continue in making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes



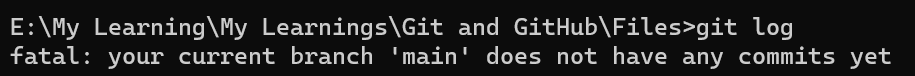
Using above will request a commit message. To ignore it, use **git commit -m <your msg>** (for all files do not include file name)

\*9122aee is a checksum or a unique string generated for each commit.

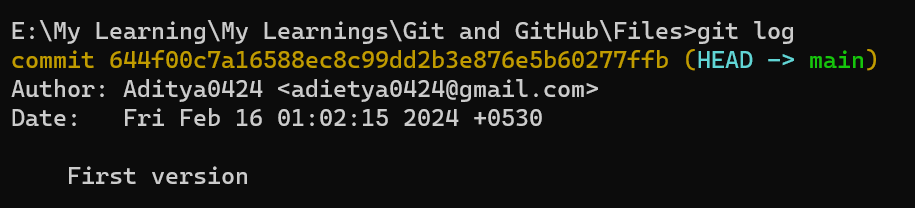
To directly commit without adding to staging area, use **git commit – a -m “your msg”**



Use git log to get list of all commits made in the project



After commit done:



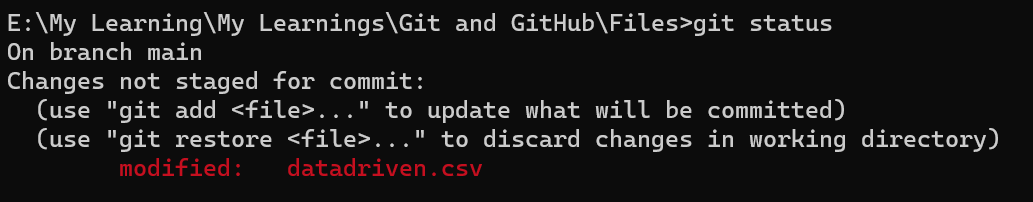
“First Version” is my commit message

You can also use git log - -oneline for a summarized list of git commits.



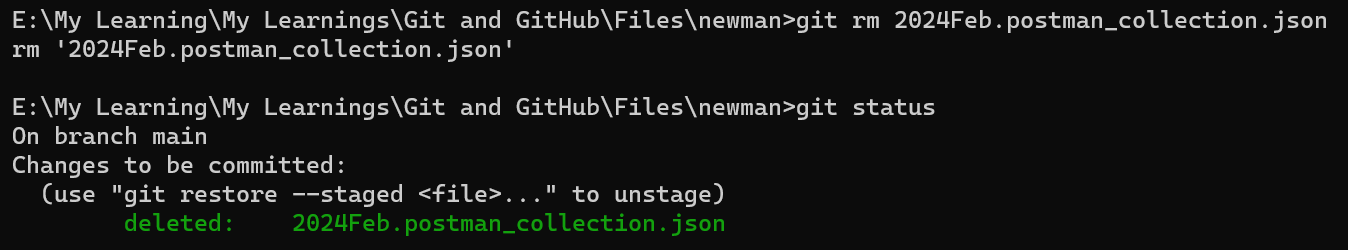
To exit git log, type “q” or “z”

If after committing, you make some changes to the file, this can be tracked again by using git status.

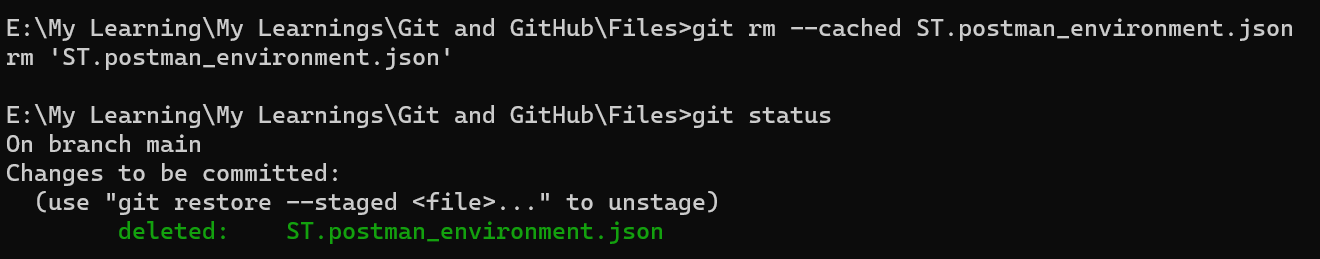


You can use git add or git restore as above.

To remove a file from local repo after committing: **git rm <file name>**

****

**You can also use git rm—cached ‘file name’ as below:**



Do git commit again to commit file deletion.

If you committed the wrong data use: **git reset <commit-id>** to reset the project to given commit id snapshot \*use **git log** to get commit ids. This operation removes all commits between the current HEAD and the specified commit. So when you do a git log, it will not show the commit in git log. Git reset retains your changes in the files.

If you want to forcefully over write your changes and move back to a complete previous state, use **git reset commitId - -hard**

Undo changes done in a commit - **git revert <commit-id>**

Diff between git reset and git revert:

Git reset will move the head pointer to previous commit. You will not see this history in git log. Lets say you wish to move back 2 commit. So if you do a git reset in this case, it will move to the previous commit as if 2 commits weren’t there.

In case of git revert, any commit will not get deleted, it will create a new commit that will exclude all changes from the commit you want to revert. So, head will be at new commit.

Step 4: Pushing changes to remote repository

In Git, you can have what we call remote repositories, or simply remotes. Remotes represent repositories that you might have read and/or write access to. Those are usually on machines other than your own, and you access them via SSH or HTTP. Remotes have names to identify them, and you can have as many remotes per repository as you need or want. However, you can't have two remotes with the same name.

One time activity to add remote repo:

**git remote add <repo alias name> <repo url>**

eg: **git remote add origin https://github.com/Aditya0424/2024June.git**

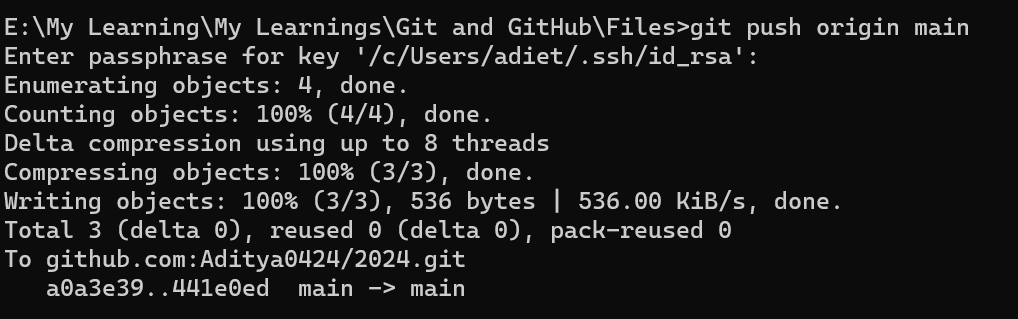
* You can use **git remote remove origin** to remove origin.
* Use git remote to get list of all remotes you’ve added
* git remote -v this will return repo names and their URLs.

****

Repository name in above case need not to be same as that in GitHub. You can use it for aliasing. For eg next time you can use “git remote add <repo alias name>”. Now since git already knows the URL to your repository, it will commit your changes there only.

To push your changes to remote repository, use **git push <aliasname> <branch name>**

**git push origin <branch name> eg: git push origin main** or to push in a specific branch to remote, use: **git push -u origin branch name**



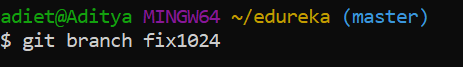
# **Branching**

Branching allows you to create independent local branches in your code.

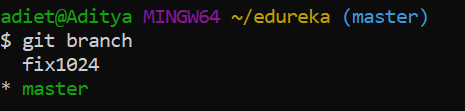
Let’s say you want to add a new feature (which is in the development phase), and you are afraid at the same time whether to make changes to your main project or not. This is where git branching comes to rescue. In the above scenario, you can create a new branch and test the new feature without affecting the main branch. Once you are done with it, you can merge the changes from new branch to the main branch.

Note: Staging area is different for different branches

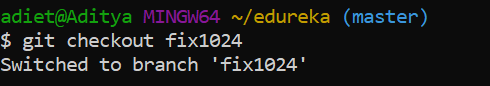
To create a new branch: git branch <branch name>

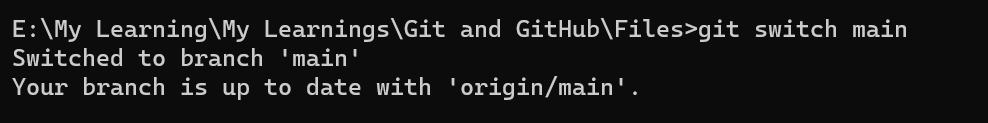


To check existing branches: git branch

\

To switch to a new branch: git checkout <branch name>

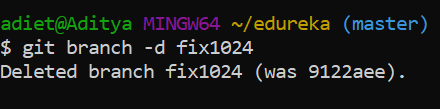


You can also use git switch to change branch

You need to commit at least one time on master before creating a new branch. To create a new branch without committing on master, you can use:

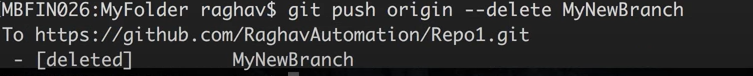
git checkout -b <branchname>

To delete a branch from local: git branch -d <branch name>



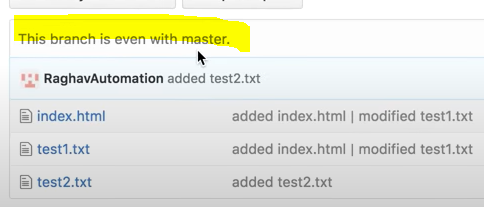
To delete a branch from remote: git push origin - - delete <branch name>

or $ git push origin -d <branchname>



To merge branch to master

1. Switch to master using “git checkout master”
2. To merge branches, use: “git merge <branch name>”
3. git push origin main – to push changes to remote

On GitHub, in your branch, you should then see below note: 

# **Setting git global properties**

git config –global user.name “<git username>”

git config –global user.email “<git email>”

to see these settings, use git config -l or git config --global -l

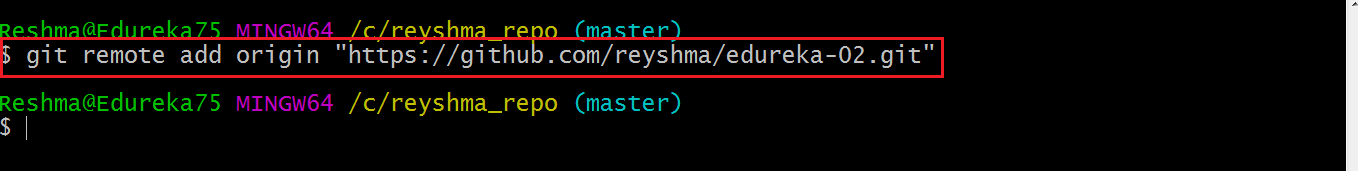
# **Clone**

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository.

# **Pull**

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances.

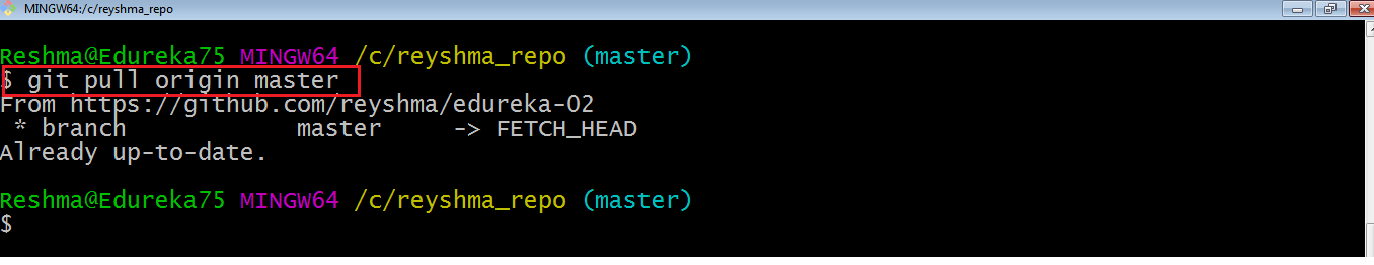
**git remote add origin <link of your central repository>**

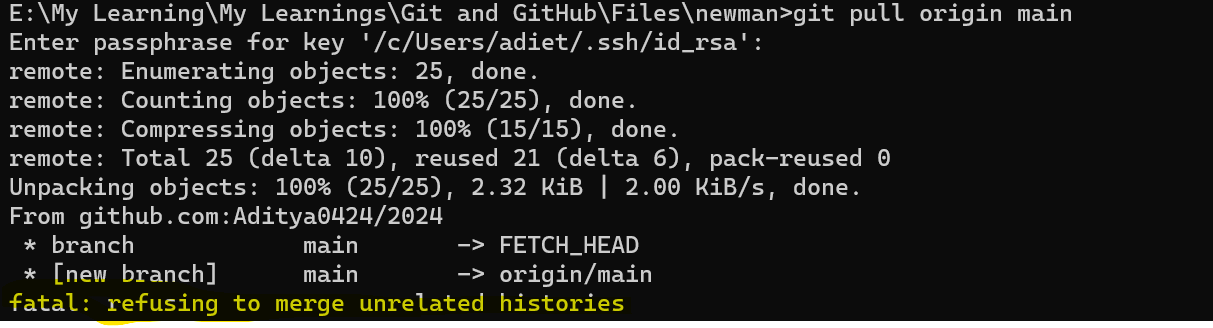


Now that my origin is set, let us extract files from the origin using pull. For that use the command:

**git pull origin master**

This command will copy all the files from the master branch of remote repository to your local repository.





git pull origin branchname --allow-unrelated-histories

# **Push**

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository.

# **Git Tag**

Tag operation allows giving meaningful names to a specific version in the repository.

When to create a Tag:

* When you want to create a release point for a stable version of your code.
* When you want to create a historical point that you can refer to reuse in the future.

# **Git Create tag**

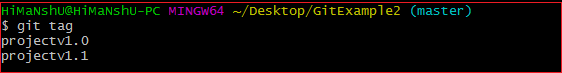
To create a tag first, checkout to the branch where you want to create a tag. To check out the branch, run the below command:

Git Tags

You can create a tag by using the **git tag** command. Create a tag with some name say v1.0, v1.1

Git Tags

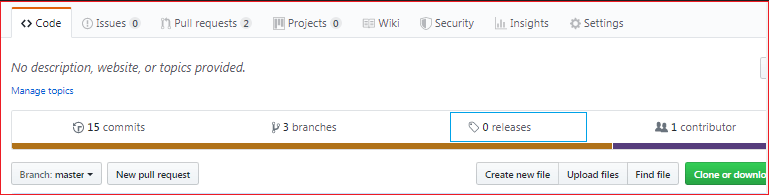
We can list the available tags in our repository. Using **git tag**



We can push tags to a remote server project. It will help other team members to know where to pick an update. It will show as release point on a remote server account. The git push command facilitates with some specific options to push tags. They are as follows:

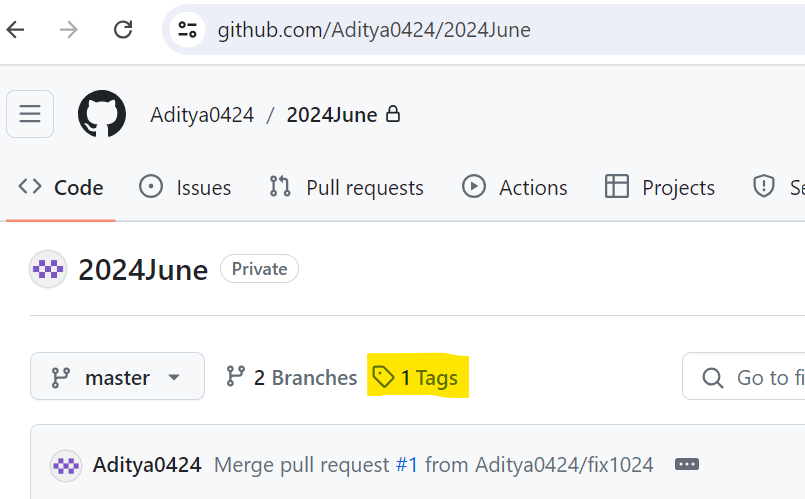
* **git push origin <tagname>**

Before



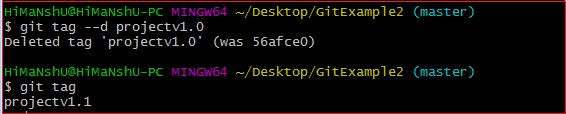
After



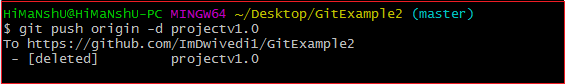


# **Git Delete Tag**

Git allows deleting a tag from the repository at any moment. To delete a tag, run the below command: $git tag -d <tagname> or $git tag --delete <tagname>



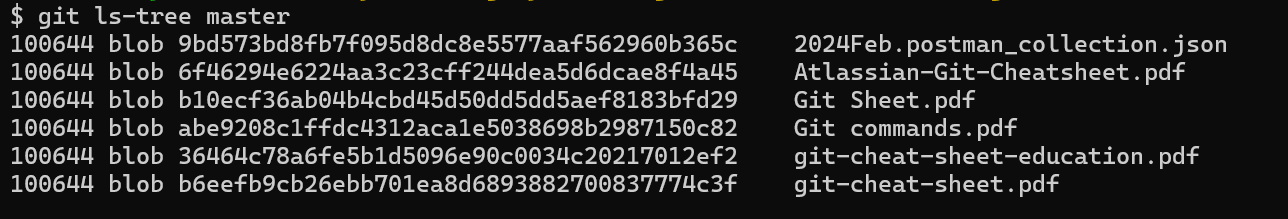
To delete a remote tag, use $ git push origin -d <tagname>



# **Get all files**

git ls-files - will print files in the current working directory.

git ls-tree master to get files in ‘master’ branch.

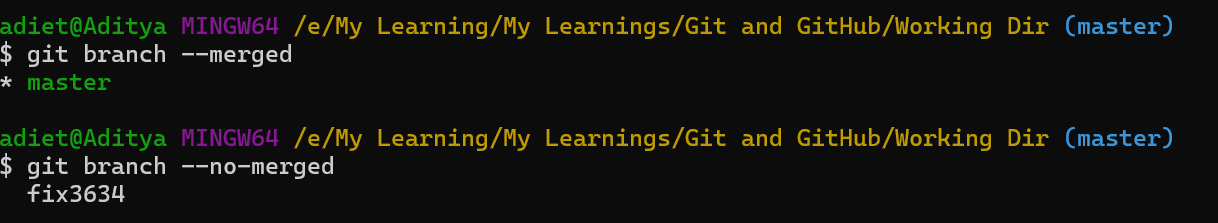


# **By what method will you know in Git if a branch has just been combined into master?**

**git branch --merged** It records the branches that have been merged into the present branch.

**git branch --no-merged** It records the branches that have not been merged.

\*\*you need to be on your main branch to run above queries

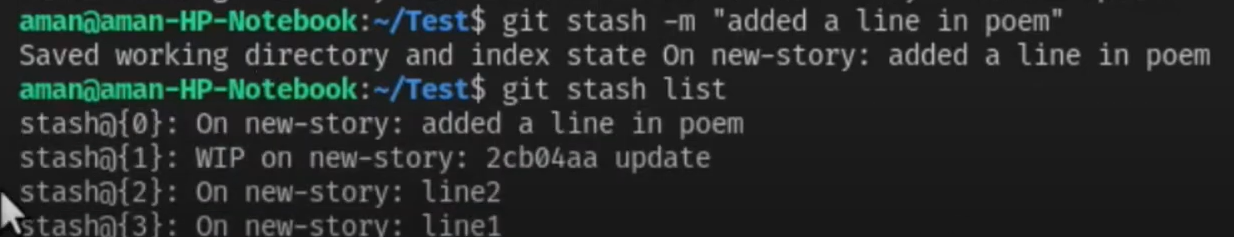


# **Git Stash**

The git stash command enables you to switch branches without committing the current branch. Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. When a developer runs the git stash command, Git stores all the changes in a stash and resets the state of the workspace to its prior commit state.

git stash -m “your message or label for your unfinished work”

use git stash list to get a list of all your git stashes



You can then use git stash apply to revert to the latest stash. To revert to any specific stash, use git stash apply stash@{2}.

stash@{2} is id given in above screenshot.

# **What is the purpose of GIT stash?**

GIT stash takes the present state of the working file and index and puts in on the stack for next and gives you back a clean working file. So in case if you are in the middle of object and require to jump over to the other task, and at the same time you don't want to lose your current edits, you can use GIT stash.

# **Difference between Git stash vs. commit**

The git stash and git commit commands are similar in that both take a snapshot of modified files in the git working tree and store that snapshot for future reference. The following shows the key differences between the two commands:

* A commit is part of the public git history; a stash is stored locally.
* A commit creates a new save point on a branch; a stash reverts to a previous save point.
* A new commit leaves files in the working tree unchanged; a stash resets files in the working tree to the previous commit point.

# **What is the difference between git pull and git fetch?**

Git pull command pulls commits from a specific branch in your central repository and updates your object branch in your local repository.

Git fetch is also used for the same objective, but it works in a slightly different method. When you use a git fetch, it pulls all new commits from the desired branch and saves it in a new branch in your local repository. If you need to reflect these changes in your target branch, git fetch should be followed with a git merge. Your target branch will only be restored after combining the target branch and fetched branch. To make it simple for you, remember the equation below: **Git pull = git fetch + git merge**

# **What is the difference between git clone and git pull?**

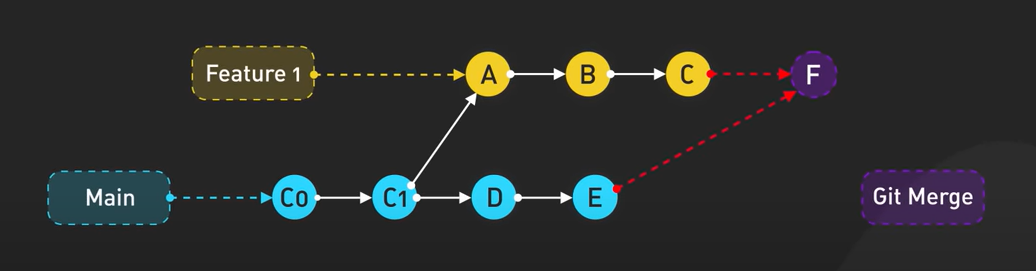
git clone is used for just downloading exactly what is currently working on the remote server repository and saving it in your machine's folder where that project is placed. Mostly it is used only when we are going to upload the project for the first time. After that pull is the better option.

git pull is a (clone(download) + merge) operation and mostly used when you update that local copy with new commits from the remote repository. If you are collaborating with others, it is a command that you will run frequently.

In laymen language we can say:

* Clone: Get a working copy of the remote repository.
* Pull: I am working on this, please get me the new changes that may be updated by others.

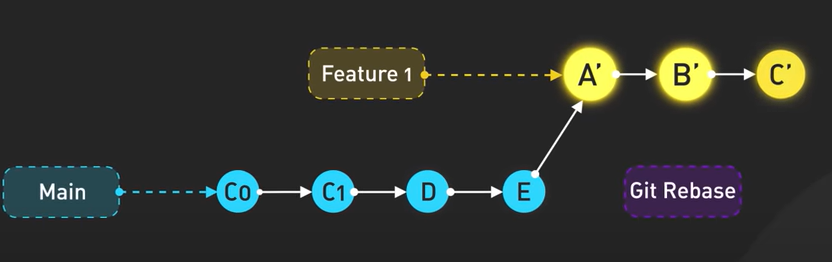
# **What is the difference between git merge and git rebase?**



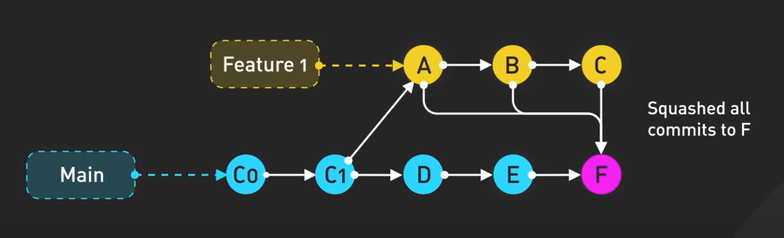
You have first created a new feature branch and added A,B and C commits. In the mean time, there were some more commits on the main branch.

Git merge here will pull all the changes from main branch and will create a new save point that includes all the changes. Now in order to merge these changes with the main branch, you’ll need to do merge again. Git merge will tie all the commits made on the main branch as well as the feature branch.

Git rebase will change the base of feature branch as per the latest commit in main branch and then we place our changes from there. It gives a clean straightforward commit history.



Another option that we have is a squash commit where we just pull in all commits in feature branch in a new savepoint (which all includes all changes to main branch). Drawback here is that we loose all commit history for feature branch.



# **What is the difference between Forking vs Cloning**

A GitHub fork is a copy of a repository (repo) that sits in your remote account. Forking is done on the GitHub Account while Cloning is done using Git. When you fork a repository, you create a remote copy of the original repository (upstream repository) but the repository remains on your GitHub account. Whereas, when you clone a repository, the repository is copied on to your local machine with the help of Git.

# **What is a pull request in Git?**

A pull request is an event in Git where a contributor asks a maintainer of a Git repository to review code they want to merge into a project. It serves as a proposal to merge changes made in one branch of a repository into another, typically from a feature branch into the main branch. Pull requests are essential for facilitating code reviews, encouraging collaboration

• Feature Additions: A developer adds a new feature, like password reset, and creates a pull request to merge it into the main branch after approval.

• Bug Fixes: A separate branch is made to fix a discovered bug, and a pull request is created for team review and integration.

• Code Refactoring: Developers refactor code for better performance or readability and submit a pull request for team approval.

Steps for PR:

* A developer creates the feature in a dedicated branch in their local repo.
* The developer pushes the branch to a remote repository.
* The developer files a pull request
* The rest of the team reviews the code, discusses it, and alters it.
* The project maintainer merges the feature into the official repository and closes the pull request.

# **What are the common Git mistakes and how to fix them?**

**Un-stage files/directories from Index**

While adding and/or modifying files you often tend to use the default behavior of ‘git add’ command, which is to add all files and directories to the Index. Many a time you feel the need to un-stage certain files or modify them one last time before committing them.

Syntax: git reset <filename/dirname>

**Forgot some changes in the last commit**

Let’s say you forgot to make some modifications and already committed your snapshot, or forgot to add a file to that last commit and also you do not want to make another commit to highlight your mistake.

Syntax: git commit –amend

**Commited wrong data**

git reset --hard <commit-id> –reset the project to given commit id snapshot

**Not Cleaning Up Old Branches**

Syntax: git branch -d branchName

**Most common: Conflicts**

A 'conflict' appears when the commit that has to be combined has some change in one place, and the current act also has a change at the same place. Git will not be easy to predict which change should take precedence. You need to use git rebase or git pull or git clone as appropriate.

**fatal: refusing to merge unrelated histories**

This error occurs when a developer attempts to combine two unrelated projects into a single branch. This issue occurs when the branch contains inconsistent commit histories and tags with the pull request and clone. To overcome this you can use git pull origin master --allow-unrelated-histories .

**failed to push some refs to**

It occurs when a developer pushes code to an external git repository. It occurs when multiple people update the same branch at once, and the remote repository has more recent changes than your local machine. To fix, git pull to your local repository.

Git best practices:

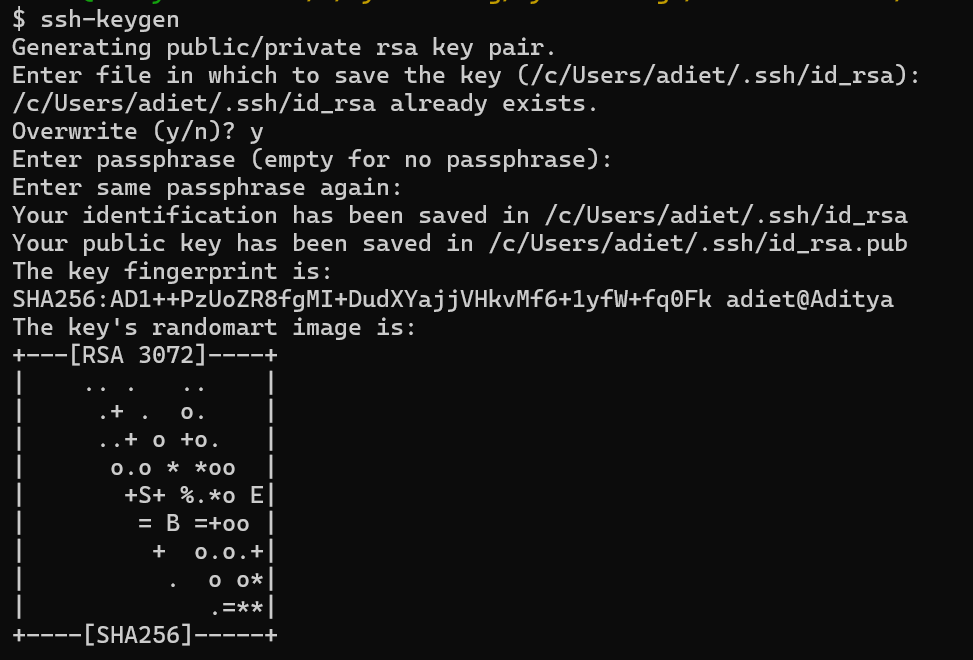
* Keep the Repository Clean and Up to Date – use git clone, git pull or git base as necessary.
* Don’t git push straight to master. Branch it out!
* Branch Frequently to Isolate Changes
* Use git diff or git status to track changes to files or file contents
* Do not commit unfinished work, use git stash to park your work.
* Keep Commits Small and Focused.
* Write descriptive and meaningful commit messages.
* Review your commit (code review)

Generate and use SSH keygen

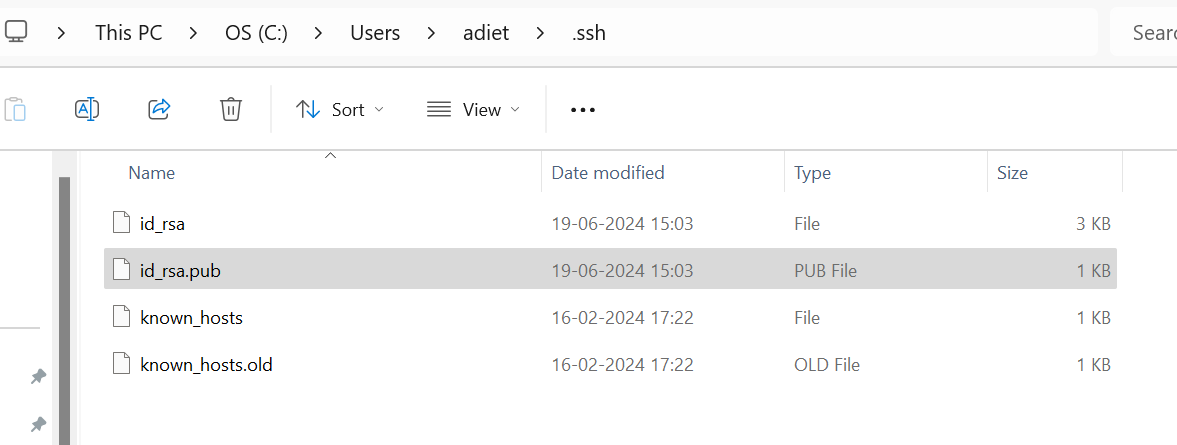
To generate SSH key type ssh-keygen

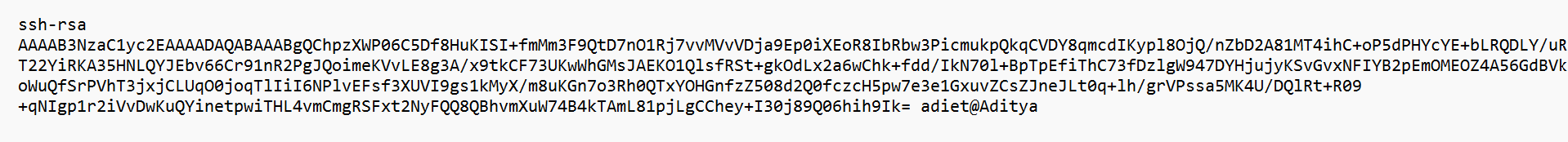
You would be prompted to enter file in which to save the key. You can press enter to save it in a default file which is under “C:\Users\adiet\.ssh”

You are then asked to set a passphrase for this key. After confirming, a key is generated.



You key can be found here





You now need to add this to your github account. To do this, go to settings>ssh & gpg keys>add new key.

From next time onwards, you’ll be asked to only enter the passphrase that you create above.

