# What is GIT

**Git** is an **open-source distributed version control system**. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

* Manage projects with **Repositories**
* **Clone** a project to work on a local copy
* Control and track changes with **Staging** and **Committing**
* **Branch** and **Merge** to allow for work on different parts and versions of a project
* **Pull** the latest version of the project to a local copy
* **Push** local updates to the main project

### **Working with Git**

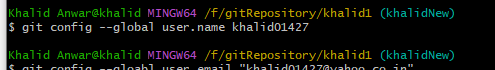
* Initialize Git on a folder, making it a **Repository**
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is **changed**, **added** or **deleted**, it is considered **modified**
* You should add modified files to a **Staging** Environment.
* The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
* Git allows you to see the full history of every commit.
* You can revert back to any previous commit.
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

# How to configure Git

Since Git is a distributed Version Control System, you will one day need to connect to other remote repositories. To avoid making any identity mistake, it is necessary to tell Git a bit about yourself.

$git config --global user.name “enterUserName” 🡪 to config username of Github account

$git config --global user.email “enterUserEmail” 🡪 to config emailid of Github account





## Checking Your Settings

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point: To quit from list, press q

Filtered List

$ git config user.name

John Doe

## Repositories

A repository is a storage where all your project and all the changes made to it are kept. You can think of it as a "change database." But don't worry; it is only a normal folder on your system, so it is very easy to manipulate.

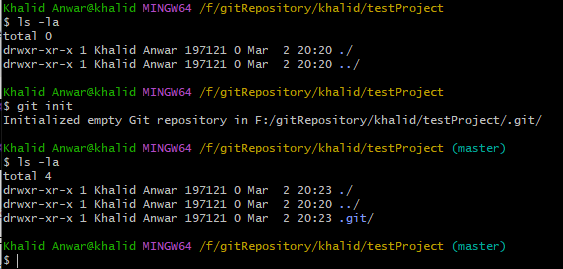
So for each project you want to start, you should

* Create the directory containing your project
* Navigate into the directory
* Initialize a Git repository

$ mkdir mynewproject

$ cd mynewproject/

$ git init

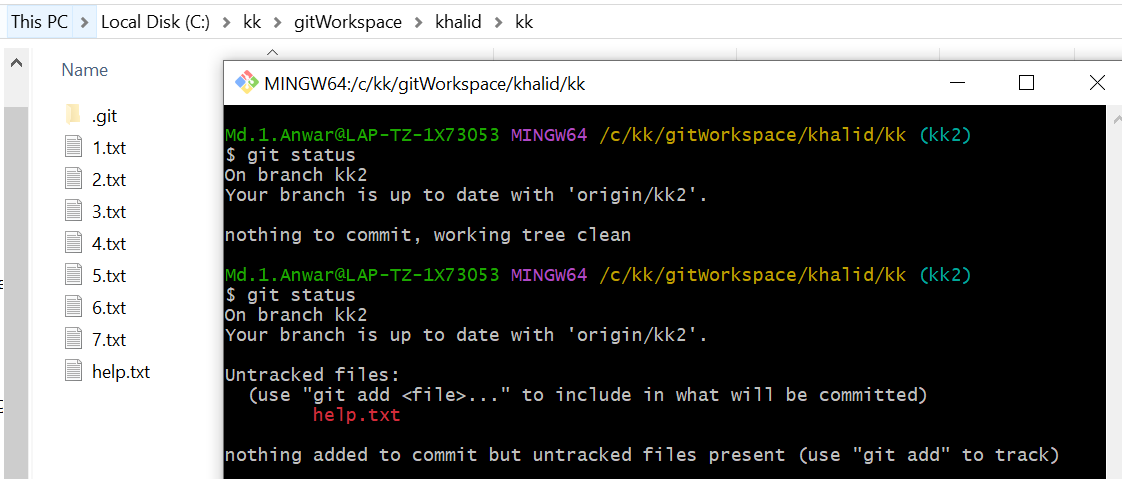


## Working Directory

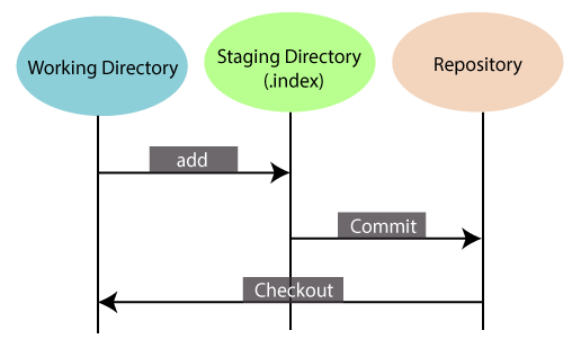
What about the empty area outside the ".git" directory? It is called the Working Directory, and the files you will be working on will be stored there. Generally, your most recent version will be on the Working Directory.

Each file you work on is on the Working Directory. There is nothing particular about this place except the fact that you will only manipulate the files here directly. Never modify the files inside the ".git" directory!

Git will detect any new file you will place in the Working Directory. And you check the status of the directory by using the Git command "status.". Like help.txt here.



## git workflow



Local directory 🡪 Staging area 🡪 github

Local changes will be first copied to staging area and then committed to github

Tracked files can be in three states:

* Modified: You changed the file.
* Staged: You changed the file and prepared it to be snapshotted.
* Committed: You took a snapshot of the entire project and the file was in it.

Remember: staging concerns only changed files you choose, while committing concerns the entire project. You stage a file; then commit the project.

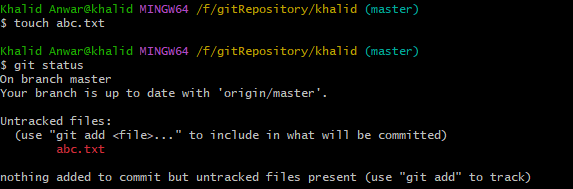
**Staging Area**

The **Staging area** is also a **unique functionality** of Git. It can be considered as a **preview of our next commit**, moreover, an **intermediate area** where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.  
Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

## Git status

List of modified files (added , deleted , changed files)

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. It allows us to see the tracked, untracked files and changes. Mostly, it is used to display the state between [**Git Add**](https://www.javatpoint.com/git-add) and [**Git commit**](https://www.javatpoint.com/git-commit) command. We can check whether the changes and files are tracked or not.



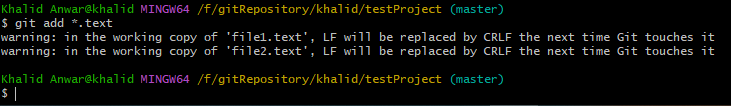
we will use the --short option to see the changes in a more compact way.

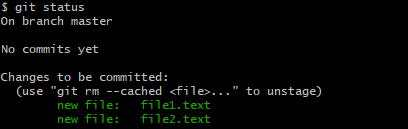
git status –short

**Note:** Short status flags are:

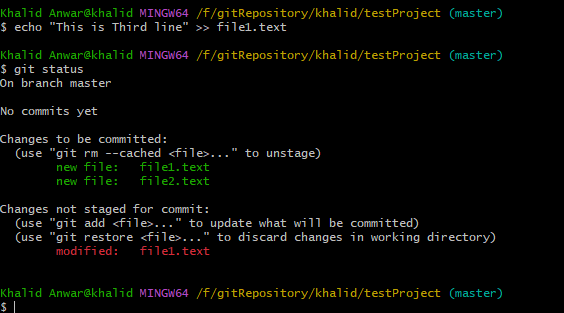
* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

## Add file





If you change file in local folder then see below



## Add more than one File

git add --all

## Delete File

You can use the [git rm](https://www.w3docs.com/learn-git/git-rm.html) command in order to delete the file from the staging area. The --cached option indicates the file to be removed from the cached area:

git rm --cached <file>

Caution. Don't forget the option "--cached" when unstaging a file. If you forget it, you could lose your file!

**commit**

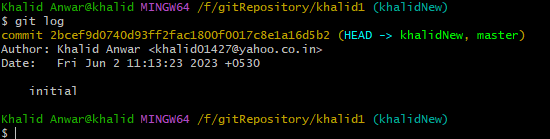
When you finish the modifications, you can commit your changes. a commit is just a snapshot of the entire project at a certain time. In addition to the snapshot, a commit also contains information about the "author" of the content and the "commiter" or who put the changeset into the repository.

git commit -m “Commiting Message”

## Git Log

To check log file(commit details) in git

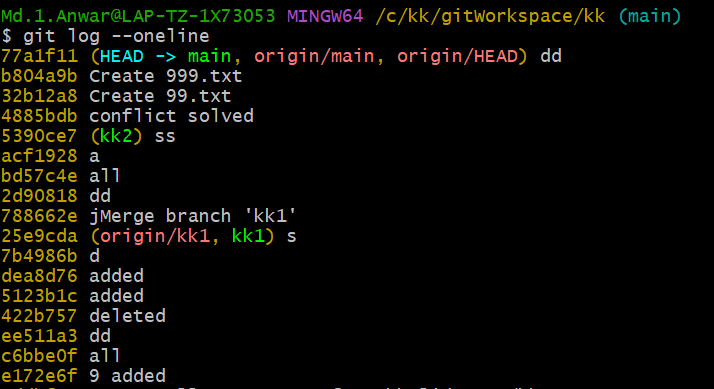
$git log



The commit log will list (from the most recent to the oldest) all the snapshots you or other people committed. If your commit history is very long, you can use the keyboard and go

* Forward or backward one line: key up and down OR j and k
* Forward or backward one window: f and b
* At the end of the log: G
* At the beginning of the log: g
* Get help: h
* Quit the log: q

you can pass the "--oneline" parameter to reduce the information. $git log –oneline

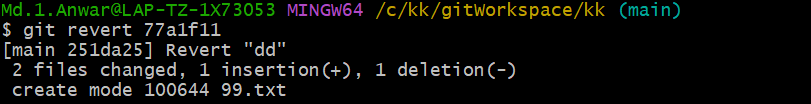


**Git revert**

The git revert command is used to apply revert operation. It is an undo type command. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit.

$git revert <Commit-HashName>

Get commit hashname. Find hashname by $git log –oneline and use this hashname.



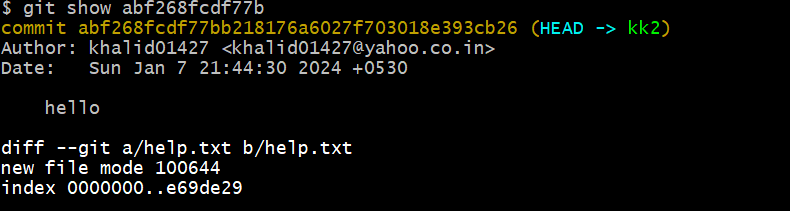
Most of time, all you want to do is reverting a single file to a previous state. This mostly happens when you've been coding for some time only to realize that your entire strategy was wrong. And instead of hitting Ctr-Z hundreds of time, it's better to revert the file.

$ git checkout -- <file>

## Git show

To show and learn what changes have been done to your project, you just use the "git show" command followed by the name of the commit. You don't even need to write the full name, just the first seven letters.

$ git show <Commit-hashName>



## Git diff

Checking previous versions is nice, but what if you only want to check the changes you just made? Checking differences between the last commit and the current working directory is an essential feature of Git. You will use it a lot! The command to check differences is simple: git diff.

git diff will show you any changes that you’ve made but not yet added to the index.) You can also get a brief summary of the situation with git status

git checkout <commitName>

When you want to bring the previous state of the project to the working directory, we have to check out the commit with "git checkout." Since this changes the files on the working directory, you have to make sure not to have any unstaged files on there. Untracked files are fine since Git doesn't track their states yet.

Caution You can't check out any other commit if your Working Directory isn't clean! Make sure to commit your changes before switching snapshots.

**Compare two branch**

$ git diff branch1..branch2

## Undo a commit

$ git revert <commit name>

# What is GitHub?

GitHub is a Git repository hosting service. GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers. It offers both **distributed version control and source code management (SCM)** functionality of Git.

# What is gui

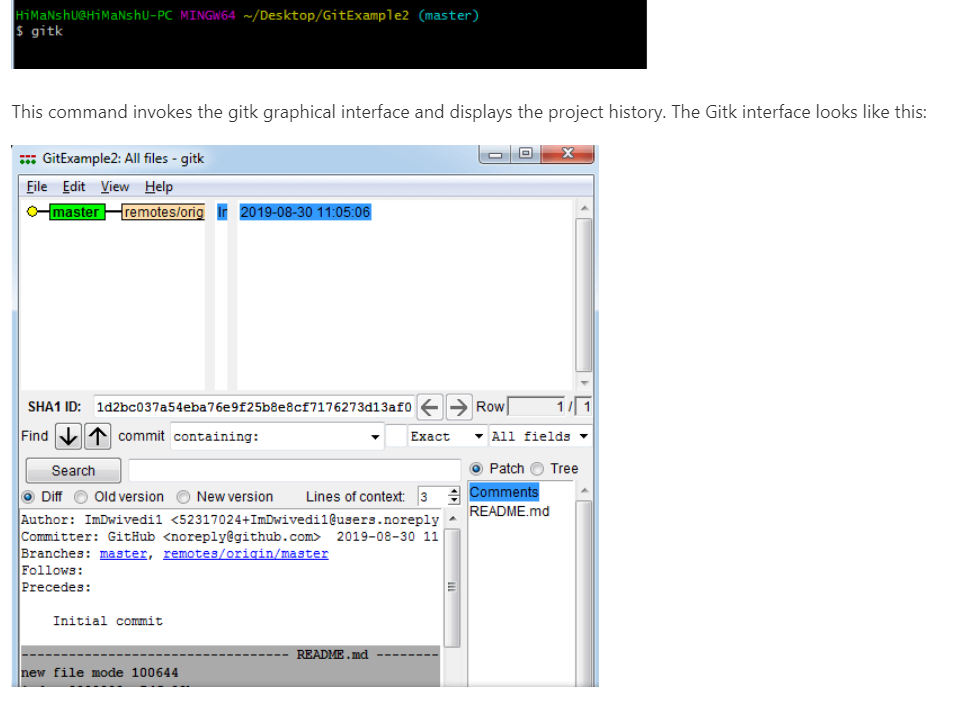
$ git gui

* Top left is a list of edited files that have not been staged yet.
* Bottom left is a list of files that have been staged.
* Top right is a diff view.
* Bottom right is a commit message text area.

# What is Gitk

gitk is a graphical history viewer tool. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.

Gitk can invoke from the command-line. Just change directory into a Git repository, and type:



## Git Third-Party Tools

Many third-party tools are available in the market to enhance the functionality of Git and provide an improved user interface. These tools are available for distinct platforms like Windows, Mac, Linux, Android, iOS.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Tools** | **Platforms** | | | | | **Price** |
| **Windows** | **Mac** | **Linux** | **Android** | **iOS** |
| SourceTree | Yes | Yes | No | No | No | Free |
| GitHub Desktop | Yes | Yes | No | No | No | Free |
| TortoiseGit | Yes | No | No | No | No | Free |
| Git Extensions | Yes | Yes | Yes | No | No | Free |
| GitKraken | Yes | Yes | Yes | No | No | Not Free |
| SmartGit | Yes | Yes | Yes | No | No | Not Free |
| Tower | Yes | Yes | No | No | No | Not Free |
| Git Up | No | Yes | No | No | No | Free |
| GitEye | Yes | Yes | Yes | No | No | Free |
| gitg | Yes | No | Yes | No | No | Free |
| Git2Go | No | No | No | No | Yes | Not Free |
| GitDrive | No | No | No | No | Yes | Not Free |
| GitFinder | No | Yes | No | No | No | Not Free |
| SnailGit | No | Yes | No | No | No | Not Free |
| Pocket Git | No | No | No | Yes | No | Not Free |
| Sublime Merge | Yes | Yes | Yes | No | No | Not Free |

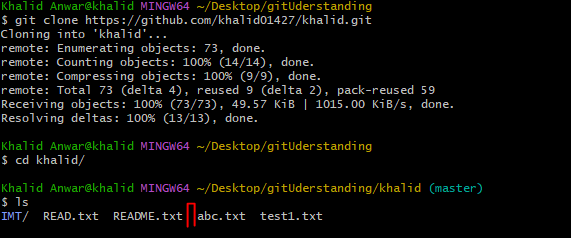
# Clone repository

<https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository>

You can clone a repository from GitHub.com to your local computer to make it easier to fix merge conflicts, add or remove files, and push larger commits. When you clone a repository, you copy the repository from GitHub.com to your local machine.

$git clone <https://github.com/khalid01427/khalid.git>

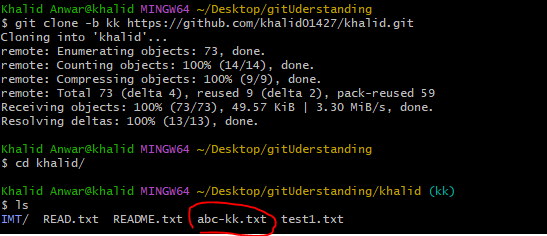
This will copy entire directory from github and put in local present working directory. This is based on active branch(master).



If we have multiple branch then clone specific branch files and folder

## Clone repository with specific branch

$git clone –b <branchName> <https://github.com/khalid01427/khalid.git>



If you want to clone files in particular folder then below syntax

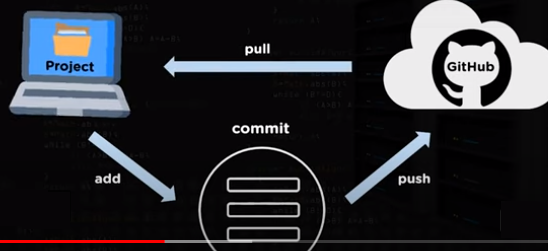
$git clone –b <branchName> <https://github.com/khalid01427/khalid.git> **FolderName**

## Clone and pull difference

git clone is how you get a local copy of an existing repository to work on. git pull (or git fetch + git merge ) is how you update that local copy with new commits from the remote repository.

## Git commit

It commits code to head(local repositories) but not to remote repositories



## Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command.

$git status

## Add file and commit

$git add filename

$git commit –m “commiting message” filename

Combining both above command

**$Git commit -a**

This command commits any files added in the repository with git add and also commits any files you've changed since then.

# Git push

Pushing means copying our local commits to a remote branch. Push action is divided into two parts too: copy of your local branch to remote and the merge of the branches. If there is no origin added then below command will have blank output



## Add origin

Add origin for remote repositories

Step1: need to setup remote **origin** using below command

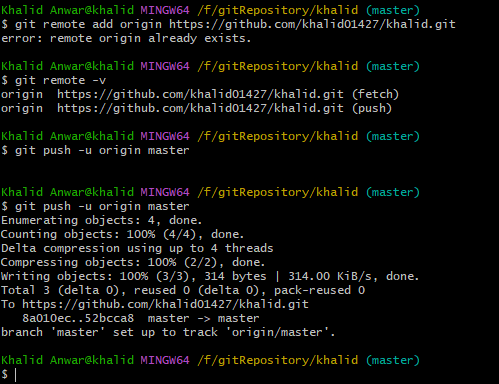
git remote add origin <https://github.com/khalid01427/khalid.git>

Step2: check remote origin setup using below command—Tracking connection

git remote –v

Step3: push commit to remote repositories(branch—master) using below command

git push -u origin master

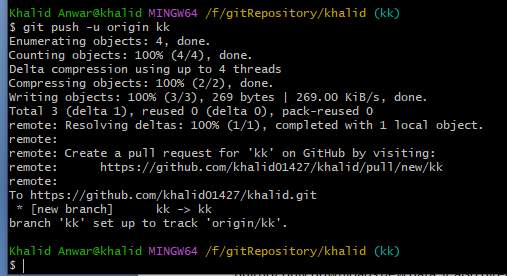
****

**Transfer commit to remote repositories**

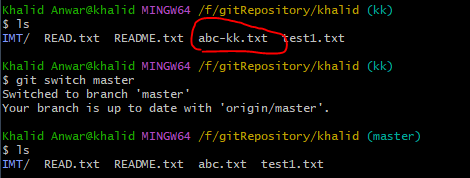
$git push –u origin master

--this will publish your changes in git hub. If your local repository is not up to date with remote github then will throw some error. You have to first make your local directory up to date.

Transferring other local branch kk to remote repositories



You can see both branches have different file list



# Git pull

In Git terminology, a pull is just the opposite of push. Push takes your branch and copies all its commits to a remote branch and creates the branch if it doesn't exist on the server yet. Pull is just that, but backward: it looks at a remote branch and copies the commits on it to your local repository. It's just an exchange of commits: push if it's from local to remote and pull if it's from remote to local.

$ git pull <remote\_name> <branch\_name>

**Fetch** + **merge**

Go and download data from github and immediately update my local repo with those changes

Fetch All

Pull (fast forward if possible)

Pull (Fast forward only)

Pull(rebase)

## How to add new file from github to local

$ git pull origin master

pull not only downloads new data; it also directly integrates it into your current working copy files. This has a couple of consequences:

Since "git pull" tries to merge remote changes with your local ones, a so-called "**merge conflict**" can occur.

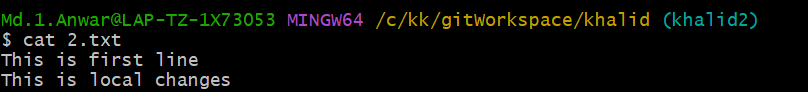
it's highly recommended to start a "git pull" only with a clean working copy. This means that you should not have any uncommitted local changes before you pull.

Use Git's Stash feature to save your local changes temporarily.

## Merge Conflict resolution

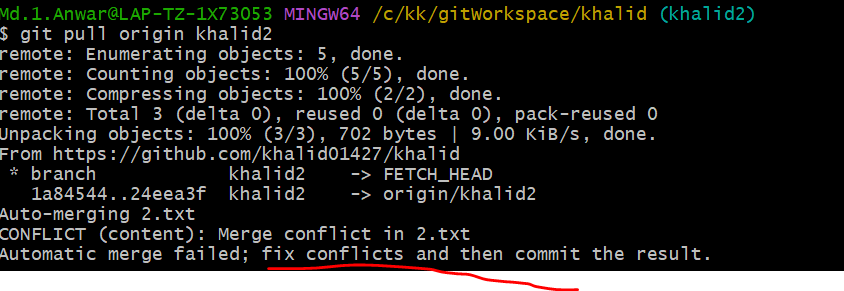
1. If remote has extra code which is not in local project
2. If local project has extra code which is not yet pushed to remote project

In above two case conflict occurs





You can see merge conflict occurs because of above case



After merge conflict, conflict file will have <<<<<<<Head which tell that this line is extra coming from remote project. , conflict file will have >>>>>>>jkj which tells that this line is extra coming from local project.

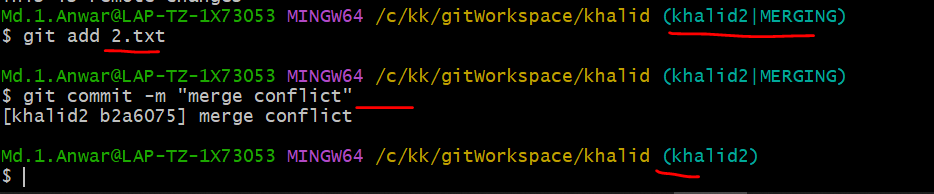
Need to resolve this conflict by modifying this file. Remove ====== , <<<<<<<<<Head and >>>>>>>jkjw and keep other changes as per other user discussion.



Add this modified file by $git add filename

Commit **all files** by @git commit –m “message content”

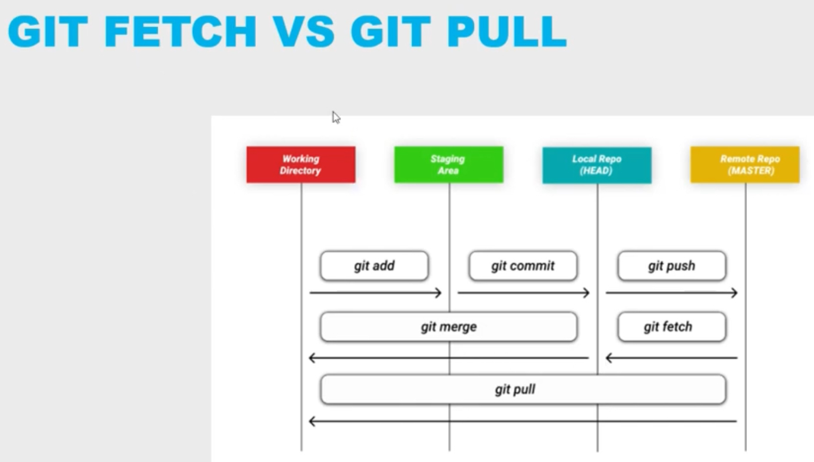
**Note**: Partial commit is not allowed and throw error



Git fetch Vs Git Pull

Git fetch only download latest changes into local repository. It downloads fresh changes that other developers have pushed to the remote repositories and allow you to review and merge manually at later time using **git merge.** Because it does not your working directory or staging area, it is entirely safe and you can run it as often as you want.

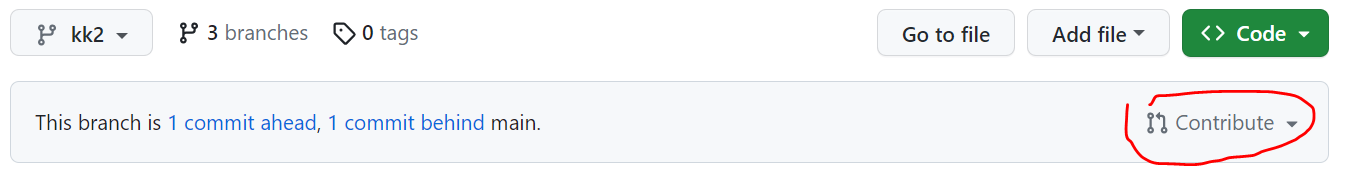
Git Pull>>downloads latet changes into local repositories and it also automatically merges changes in your working directories. It does not give you chances to review the changes before merging and as a consequences, **merge conflicts** can and do occur. One important thing to keep in mind is that it will merge only into the current working directory and other branches will stay unaffected.

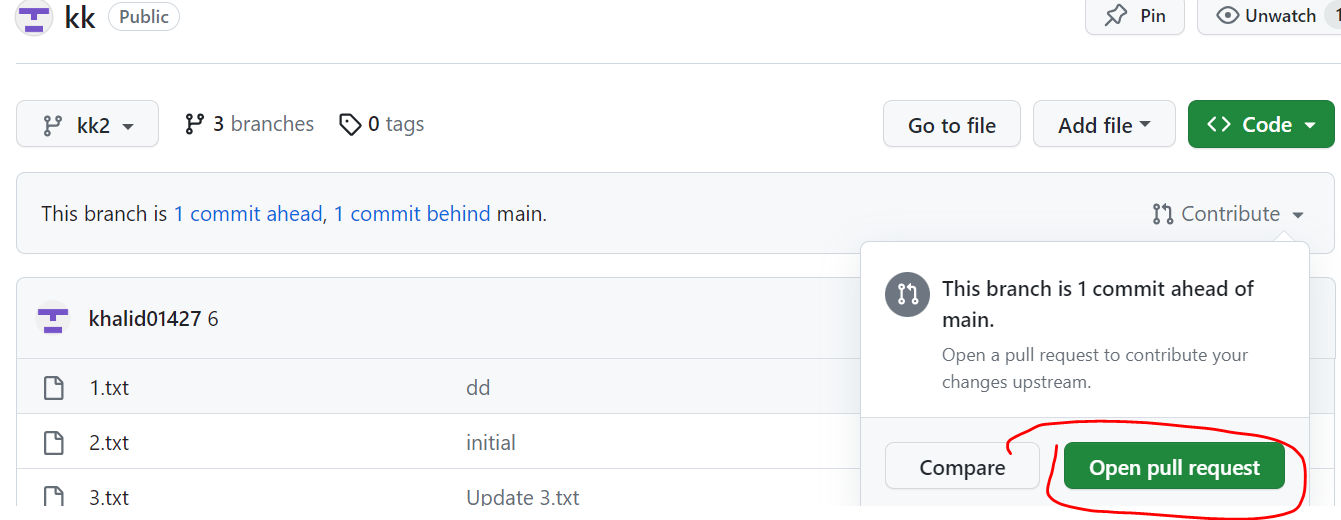


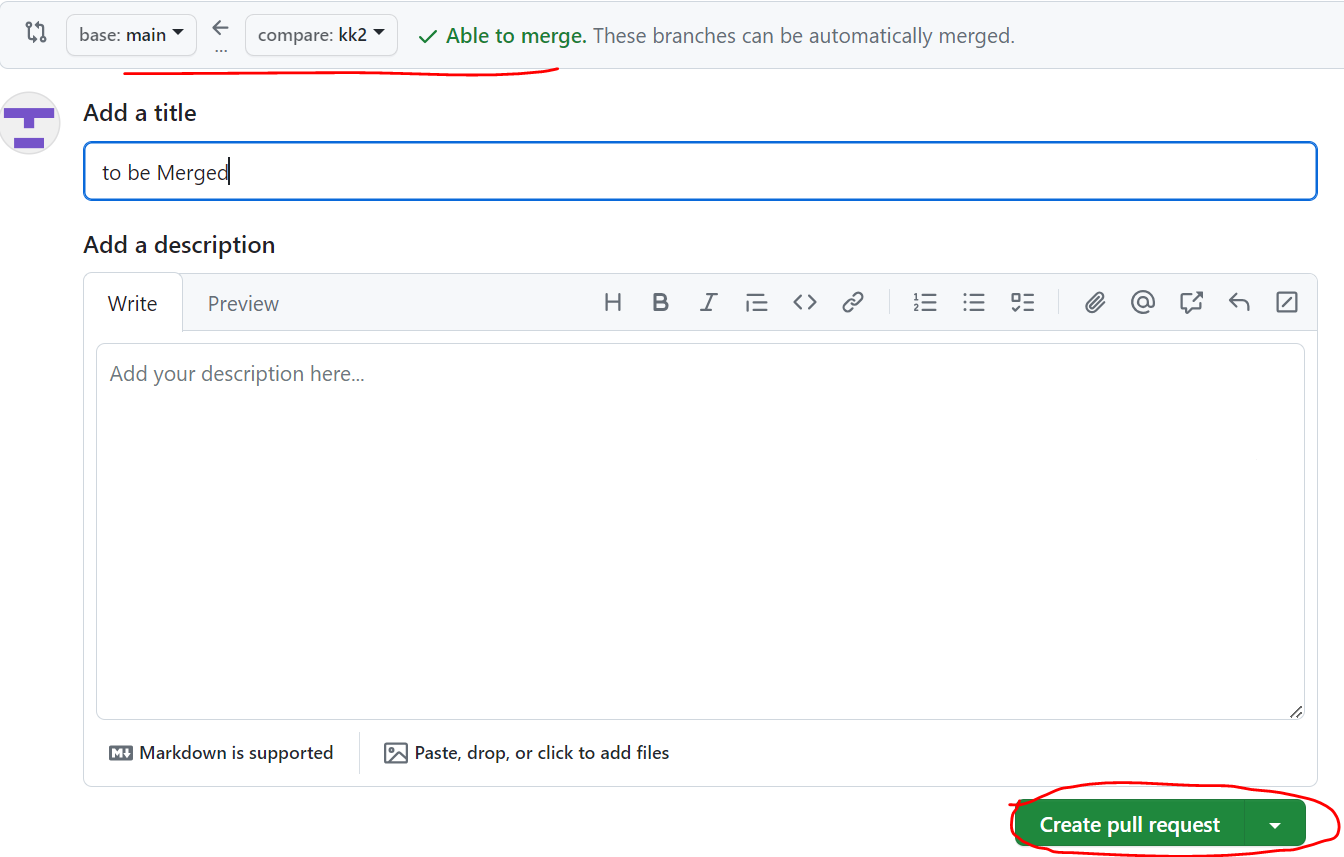
# Merging development Branch to main Branch in github

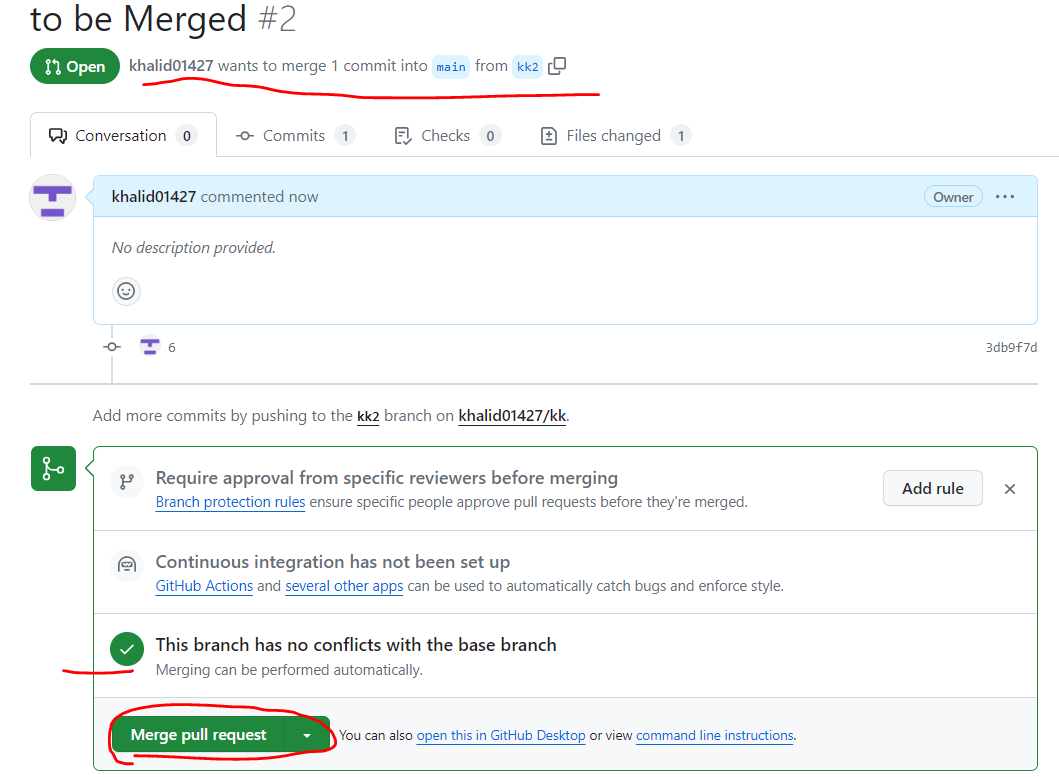
Select branch whose code to be merged to master branch

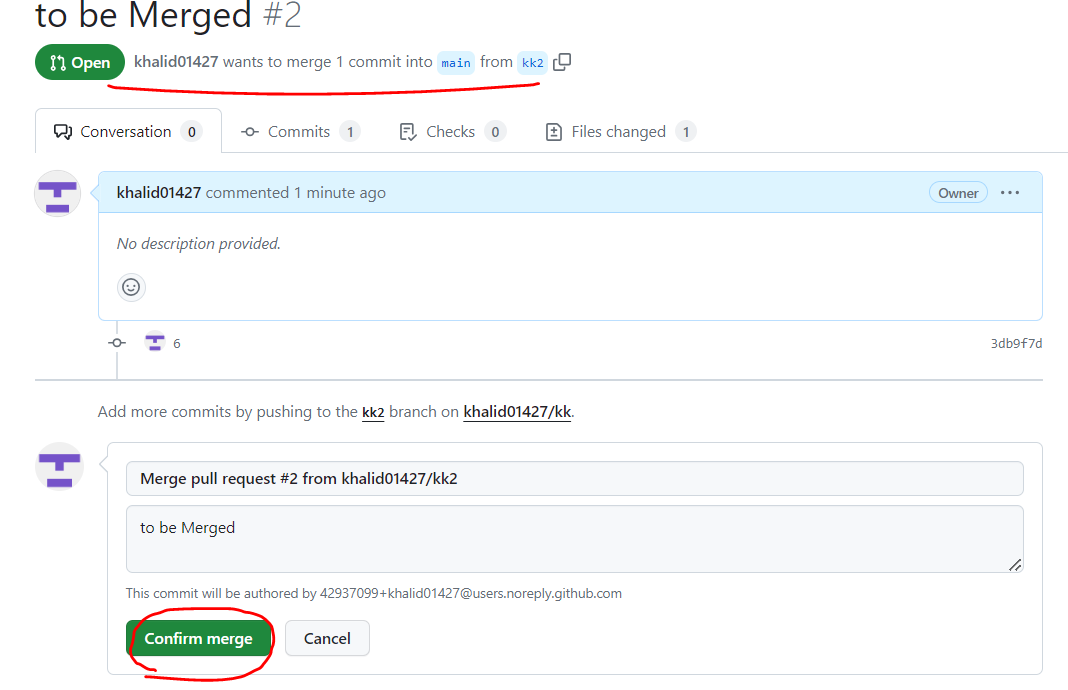
Click on contribute link

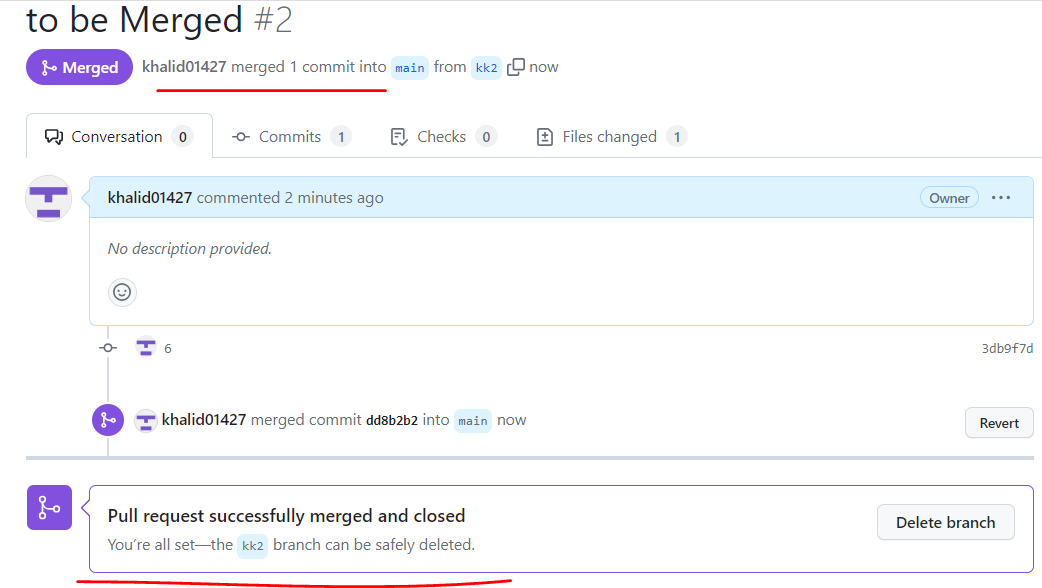






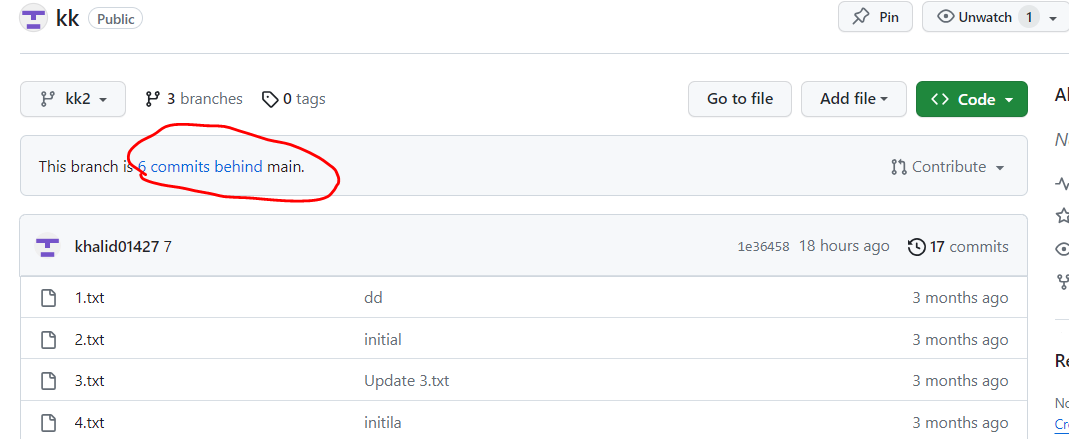


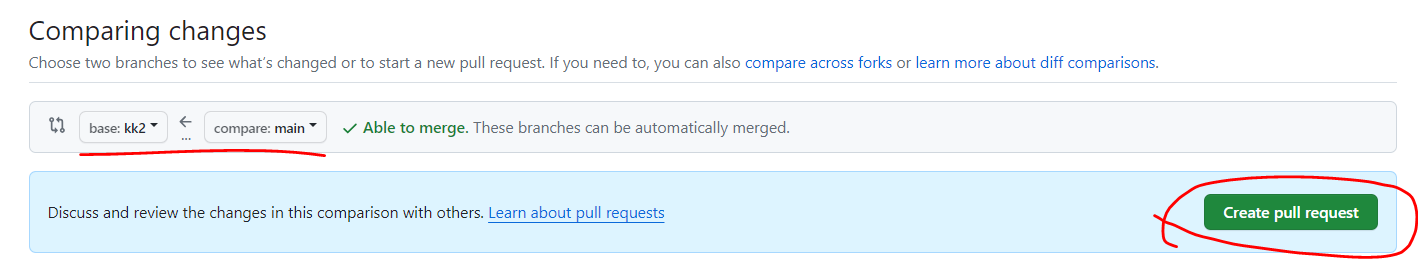


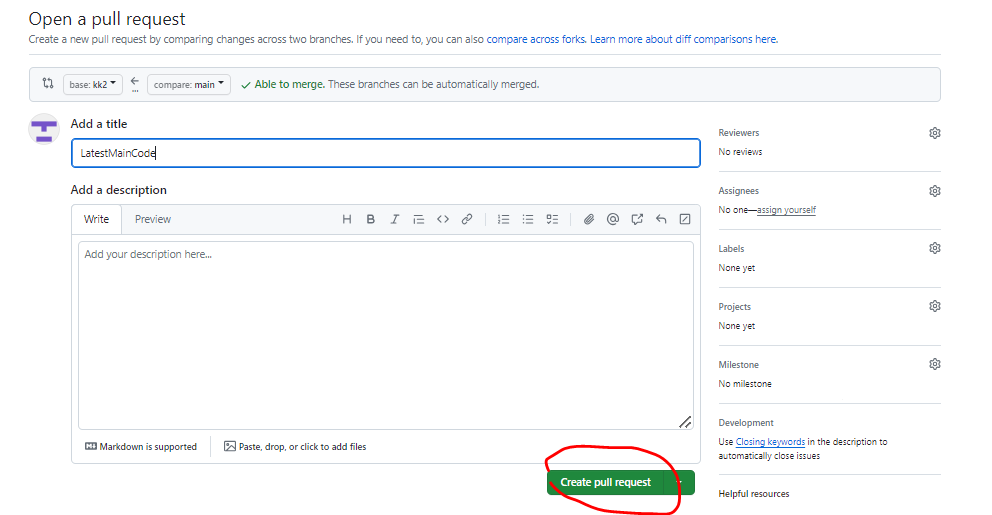


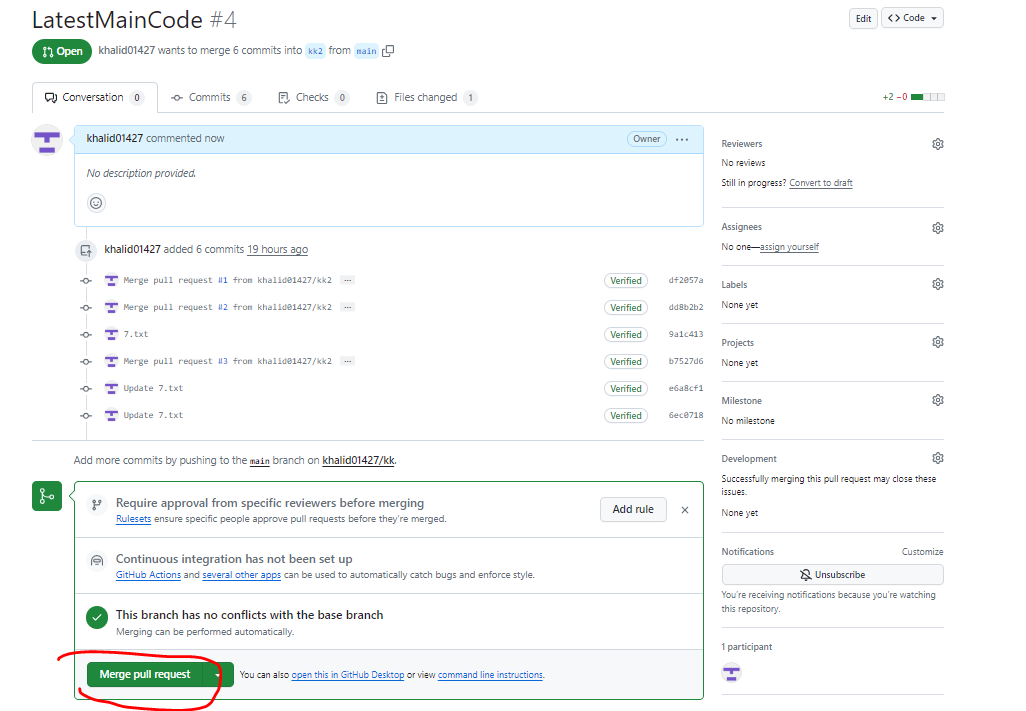
# Pull updated code from main to dev branch

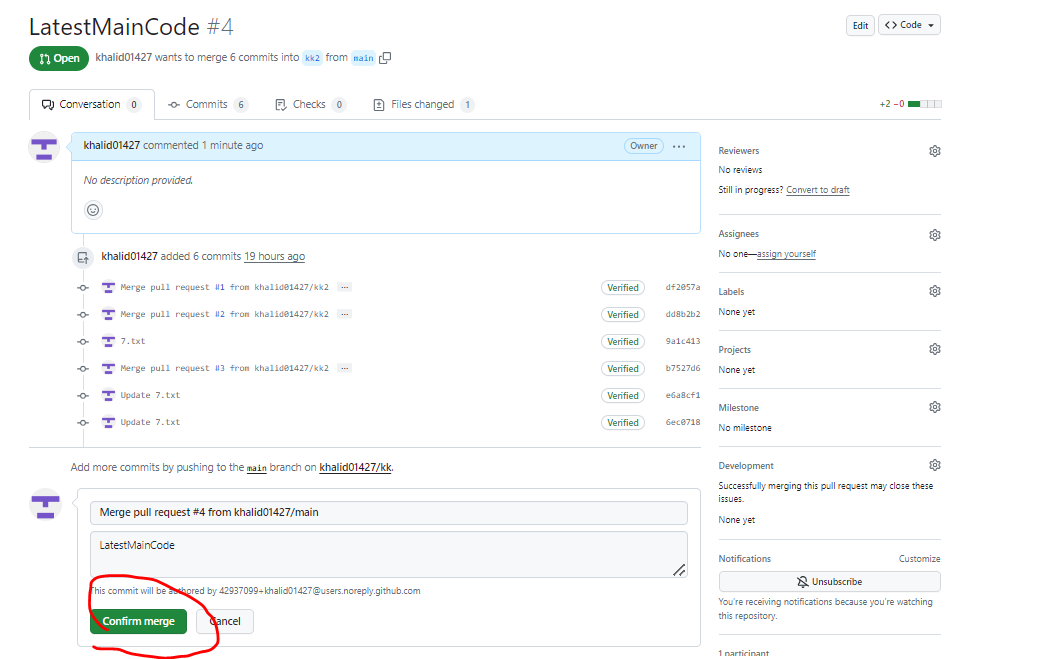
Click on “\*commit behind”

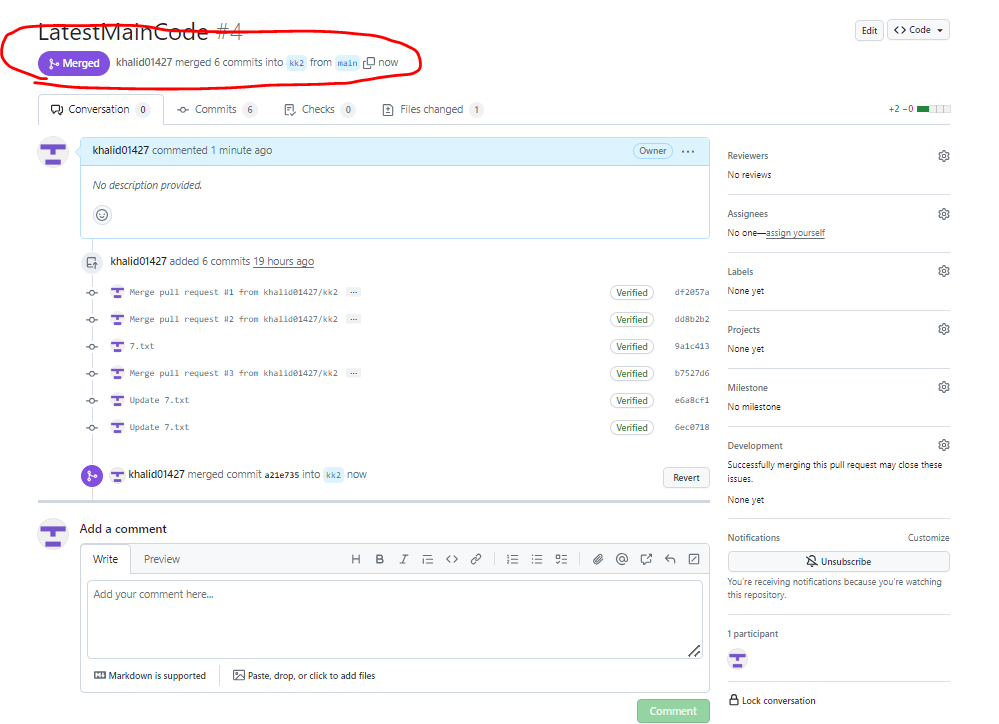








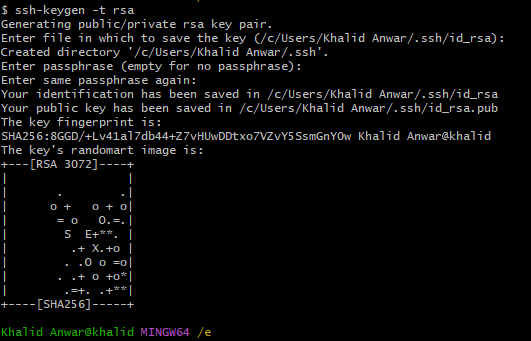




# Generate rsa token

$ ssh-keygen –t rsa

Press enter multiple times till you see output as below

****

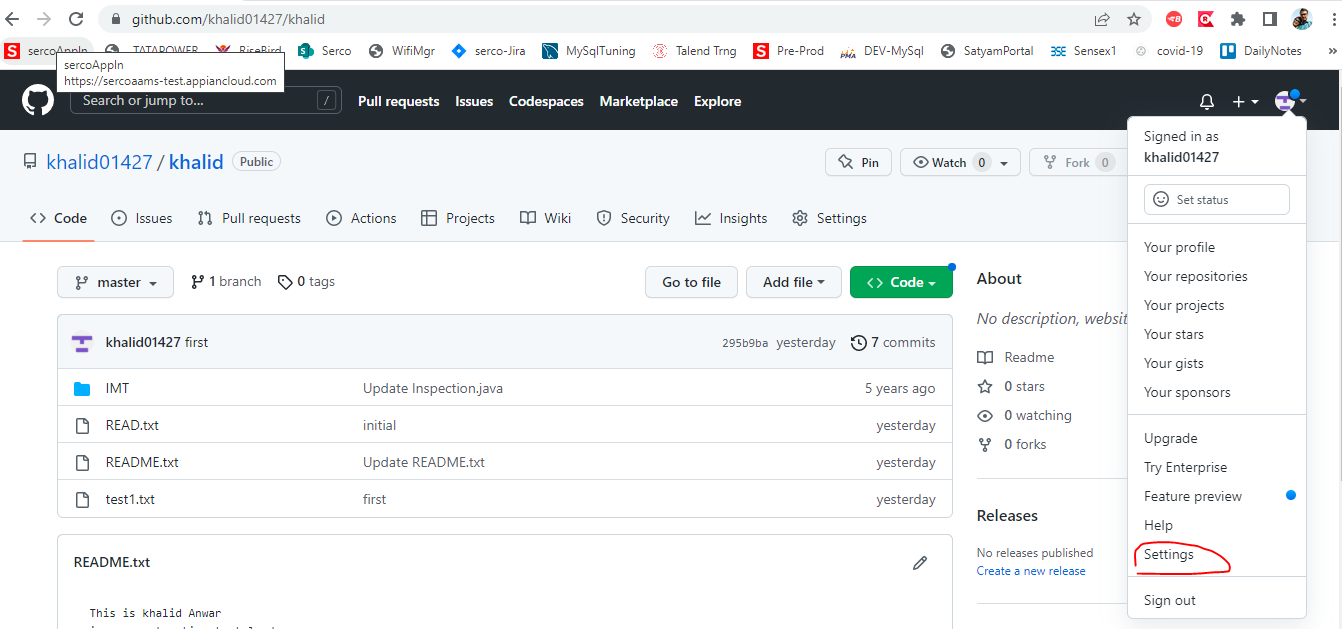
Public key would be saved in file id\_rsa.pub at c:\users\userName

Open this file and copy content to clipboard alternatively

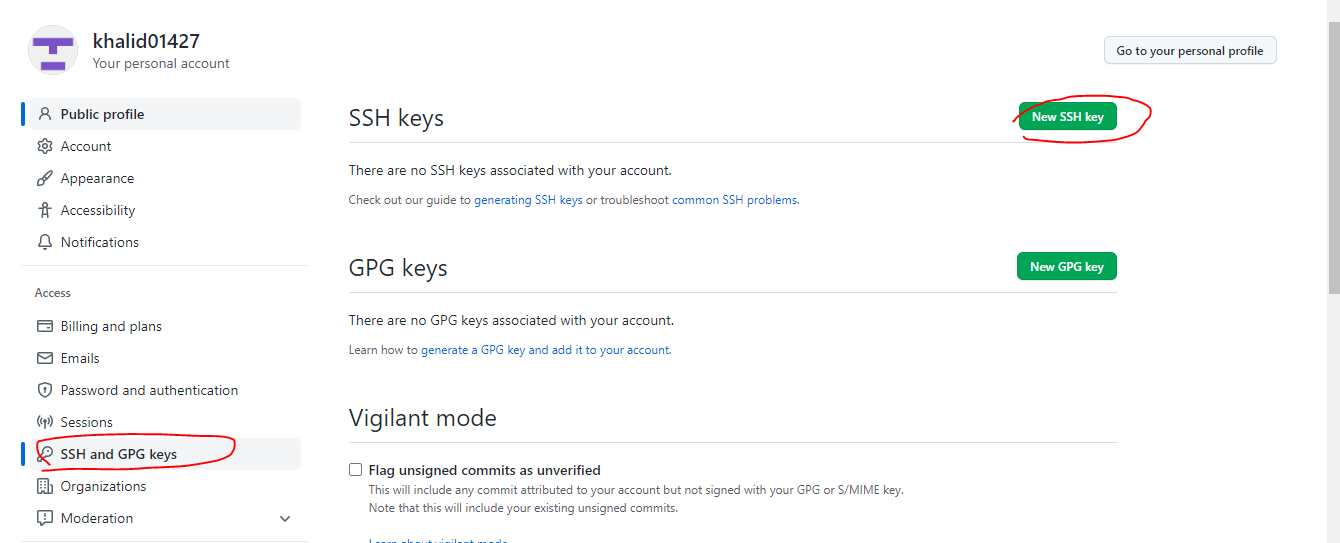
$clip < ./ssh/id\_rsa.pub

Step2:

Open Github site >> Go to profile setting



Select “SSH and GPG Keys”



Enter friendly title and paste public key

# Stashing

## Saving Changes Temporarily

A commit wraps up changes and saves them permanently in the repository. However, in your day-to-day work, there are a lot of situations where you only want to save your local changes temporarily. For example, imagine you're in the middle of some changes for feature X when an important bug report comes in. Your local changes don't belong to the bugfix you're going to make. You have to get rid of them (temporarily, without losing them!) and continue working on them later.

Situations like this one happen all the time: you have some local changes in your working copy that you can't commit right now - and you want or need to start working on something else. To get these changes out of your way and have a "clean" working copy, Git's "Stash" feature comes in handy.

### The Stash

Think of the Stash as a clipboard on steroids: it takes all the changes in your working copy and saves them for you on a new clipboard. You're left with a clean working copy, i.e. you have no more local changes.

$ git stash push

Later, at any time, you can restore the changes from that clipboard in your working copy - and continue working where you left off.

You can create as many Stashes as you want - you're not limited to storing only one.

**Note**

Just using the command "**git stash**" is the same as using "**git stash push**" It's recommended to use the full command because it's more intuitive and easier to understand.

$ git stash list -- listing all stashes

The newest Stash will always be at the top of the list, named "stash@{0}". Older Stashes have higher numbers.

$git stash pop --will apply the newest Stash and clear it from your Stash clipboard.

~~$git stash apply <stashname> --will also apply the specified Stash, but it will remain saved.~~ You can delete it later via $git stash drop <stashname>.

### **When to Stash**

Stashing helps you get a clean working copy. While this can be helpful in many situations, it's strongly recommended...

* **before checking out a different branch.**
* **before pulling remote changes.**
* **before merging or rebasing a branc**h

Example

Many times, you will want to navigate between branches but can't because your Working Directory is dirty. In this context, dirty means that you have uncommitted changed files, be they in modified or staged state. The only way to change branch is to first commit them. But most of the time, you won't be ready to commit yet because the issue at hand is not resolved yet.

# Resetting

I hope you won't use this feature often because it's very destructive! Sometimes, you want to discard everything you've done and work on a clean plate, even if you've already committed your project.

The git reset command is used to undo the changes in your working directory and get back to a specific commit while discarding all the commits made after that one.

git reset –soft <commit ID>

git reset –mixed <commit ID>

git reset –hard <commit ID> moves back to the head with the <commit ID>

# Git Branch

A single Git repository can maintain multiple branches of development.

<https://www.w3schools.com/git/git_branch.asp?remote=github>

<https://www.youtube.com/watch?v=e2IbNHi4uCI>

In Git, a branch is a new/separate version of the main repository. Let's say you have a large project, and you need to update the design on it. With Git:

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix
* Fix the unrelated error and merge the small-error-fix branch with the main branch
* You go back to the new-design branch, and finish the work there
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

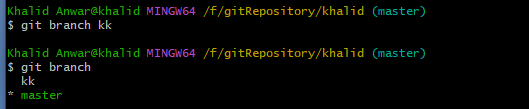
When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

## Create new branch

$git branch <new-branch-name>

Keep in mind that the branch name should only contain alphanumeric values and dashes or underscores; no spaces allowed.Let's confirm that we have created a new branch



kk--- it has been created as local branch

After you execute that command, you will notice that nothing has changed in your project. That's because creating a branch is just about creating a reference to the last commit of the current branch and nothing else. To begin working with a branch, you have to switch to it.

## View branch

Branches which are available in github, bitbucket are remote branch

$git branch –a

List out all branches (local and remote)



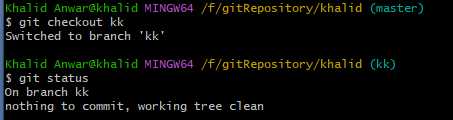
## Switching Branch

$git **checkout** kk

Or

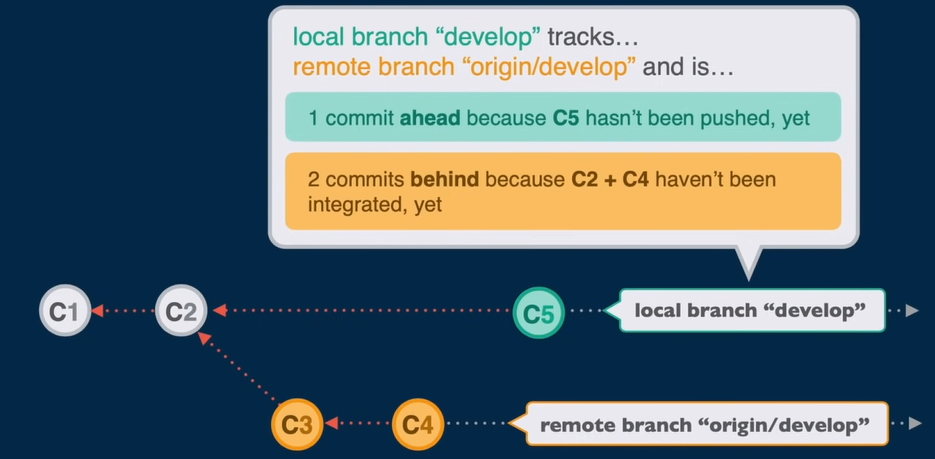
$git switch master ---better option

Switching from master to kk is called checkout



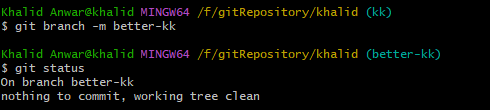
To immediately switch to a new branch after creating it, use the option "-b" with the git checkout command. For example, "git checkout -b testing" is the same as "git branch testing" and then "git checkout testing"

Note:Like when we navigated between versions, you can't switch branches if you have uncommitted changed files. Commit before you move. Or use a technique called "stashing" that we will see in later chapters.



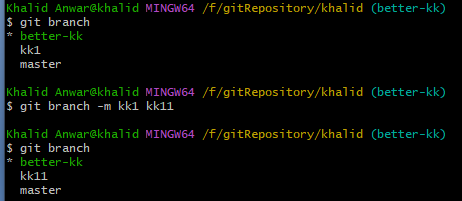
## Renaming Branch

This will rename to head branch(branch which is currently active)



Renaming non-head branch by

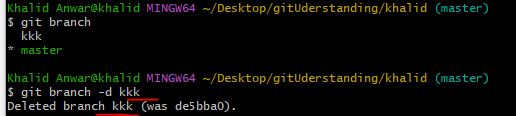
$git branch –m <existing-branch> <to-be-renamed-branch>

****

## Deleting branch

You can not delete branch which is active.

$git branch –d <branchNametoBeDeleted>



Just like a real tree branch, you don't cut the Git branch you are currently standing on. Check out another branch before deleting the branch

## Deleting remote branch

$git push origin --delete <branchName>

## Merging Branch

$git merge <branchName> wherein changed file will be copied to active branch.  Merging is the act of combining two branches or, more correctly, pouring a branch into another. Branches can be formed from any other branch, and when a branch has been created, it becomes independent from its parent. Changes done to either branch won't affect the other, until it's time to merge.

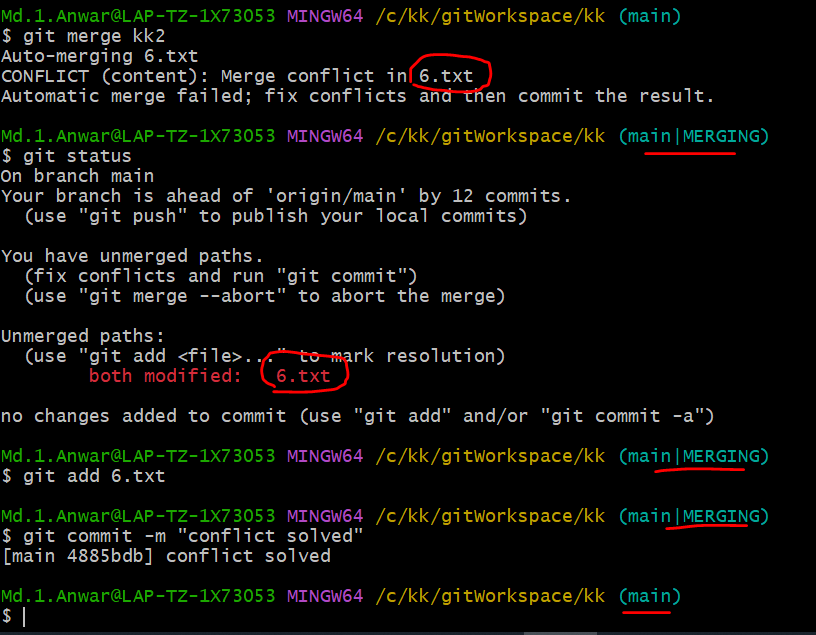
As shown below, changes of branch “**khalid2a**” will be merged (copied) to active branch “**khalid2**”

Note: Merging does not mean that all files of khalid2 are same in khalid2a



If the changes don’t conflict, you’re done. If there are conflicts, markers will be left in the problematic files showing the conflict;

Conflict example



**Note**: Partial commit is not allowed and throw error

### Git merge –abort

Conflict occurs While performing pull request then merging can be aborted by above command

### Understand Merging

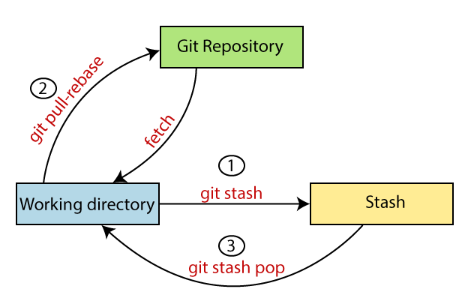
Let's imagine a situation where you create a child branch and made commits on that new branch. When the time to merge comes, several situations can arise.

If the parent branch didn't change (no commits were made) and you attempt to merge, a "fast-forward" merge will occur. A "**fast-forward**" merge is technically not a merge but just a reference change in Git. Remember that Git commits behave like chained lists, meaning that a commit contains a reference to the previous one. In fact, if the parent hasn't changed, Git just moves the reference to the parent forward (following the chained list), and the last commit in the child branch becomes the last commit of the parent branch. To put it simply, Git just appends the commits in the child branch to the parent branch. This is the easiest type of "merge" but also the most uncommon unless you work alone.

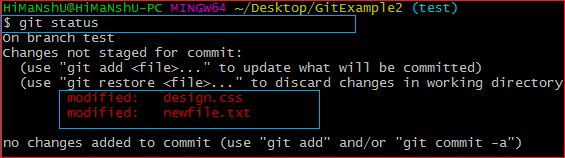
In contrast, if the parent branch has been changed (received commits), a fast-forward merge is not possible. What will occur is called a "true merge" or a "three-way merge." This is the type of merge that we've seen last chapter. This type of merge will create a new commit that has all the changes in the child branch and append that commit to the parent branch. This commit is called a "**merge commit**," and it has two parents: the parent and the child branches. If different commits from the parent and the child branches modified the same line of code, a conflict arises, and the developer must manually choose which changes to keep.

# Git Stash

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. The **git stash command** enables you to switch branches without committing the current branch. Git temporarily saves your data safely without committing.



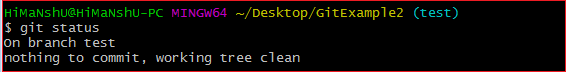
Let's understand it with a real-time scenario. I have made changes to my project GitExample2 in two files from two distinct branches. I am in a messy state, and I have not entirely edited any file yet. So I want to save it temporarily for future use. We can stash it to save as its current status.



From the above output, you can see the status that there are two untracked file **design.css** and **newfile.txt** available in the repository. To save it temporarily, we can use the git stash command. The git stash command is used as:



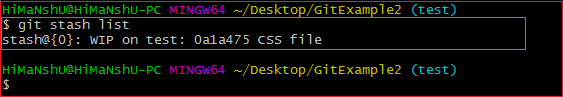
In the given output, the work is saved with git stash command. We can check the status of the repository.



As you can see, my work is just stashed in its current position. Now, the directory is cleaned. At this point, you can switch between branches and work on them.

**Git Stash List**

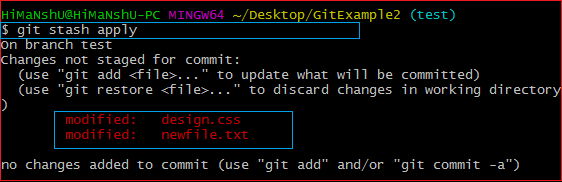
To check the stored stashes, run the below command:



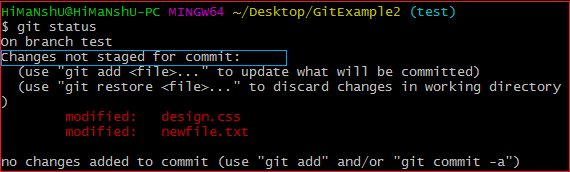
In the above case, I have made one stash, which is displayed as "**stash@{0}: WIP on the test: 0a1a475 CSS file**". It will show all the stashes with indexing as stash@{0}: stash@{1}: and so on.

## Git Stash Apply

You can re-apply the changes that you just stashed by using the git stash command. To apply the commit, use the git stash command, followed by the apply option. It is used as:



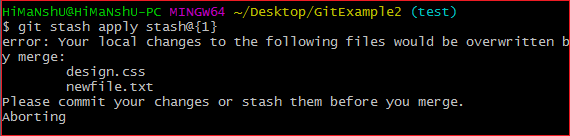
The above output restores the last stash. Now, if you will check the status of the repository, it will show the changes that are made on the file. Consider the below **output:**



From the above output, you can see that the repository is restored to its previous state before stash. It is showing output as "**Changes not staged for commit**."

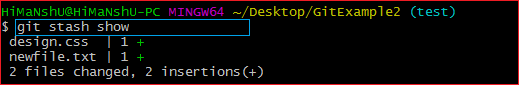
## Git Stash Apply <stash id>

In case of more than one stash, you can use "git stash apply" command followed by stash index id to apply the particular commit. If we don't specify a stash, Git takes the most recent stash and tries to apply it.



## git stash show

The above command will show the file that is stashed and changes made on them. Consider the below output: The below output illustrates that there are two files that are stashed, and two insertions performed on them.



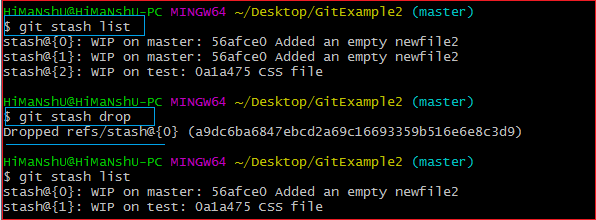
## git stash pop

The git stash pop command is quite similar to git stash apply. The main difference between both of these commands is stash pop command that deletes the stash from the stack after it is applied.

## git stash drop

The **git stash drop** command is used to delete a stash from the queue. Generally, it deletes the most recent stash. Caution should be taken before using stash drop command, as it is difficult to undo if once applied.

The only way to revert it is if you do not close the terminal after deleting the stash. The stash drop command will be used as:



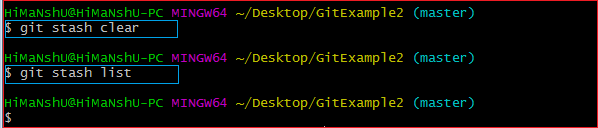
In the above output, the most recent stash **(stash@{0})** has been dropped from given three stashes. The stash list command lists all the available stashes in the queue.

git stash drop **<stash** id**>**   --Delete specific stash

$ git stash drop stash@{1}

## Git Stash Clear

The **git stash clear** command allows deleting all the available stashes at once. To delete all the available stashes, operate below command:



# Ignore File

## What Is a .gitignore File?

A .gitignore file is a plain text file that contains a list of all the specified files and folders from the project that Git should ignore and not track.

Inside .gitignore, you can tell Git to ignore only a single file or a single folder by mentioning the name or pattern of that specific file or folder. You can also tell Git to ignore multiple files or folders using the same method.

You can create a .gitignore file in your repository's root directory to tell Git which files and directories to ignore when you make a commit.

# Create new branch in Github with new project

Step1: Create folder name

Step2: got to new folder name

Step3: initilaze this folder

Step4. Create one dummy file



Step5. Add this file and commit to master branch

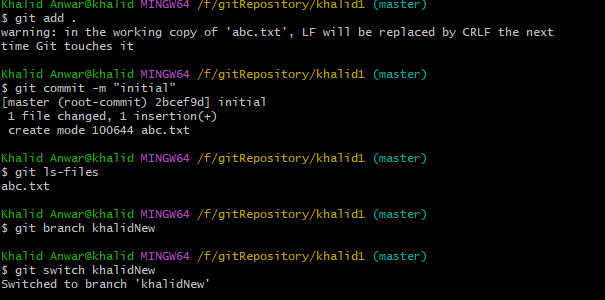
Step6: create new branch name “khalidNew”

Step7: switch to new branch

Note: (step5,6 7 can be skipped by)

$git switch --orphan khalidNew

$git commit --allow-empty --m “initial commit”



Step8: set user name and mail in configuration file

Step9: add remote origin





Step10: finally push code to new branch

Initially “khalidNew” branch is not created in Github but when we push code then this will create new branch and available commited code will be pushed to github server



## Getting a Git Repository

You typically obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository $git init

Above command will create .git folder with many sub folder inside .get folder

OR

1. You can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

git stash

git pull

git stash pop

git commit -m"this is new"

git push

## Git Command

$ git --version -->this gives git version installed in your system

# Install Git bash

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

