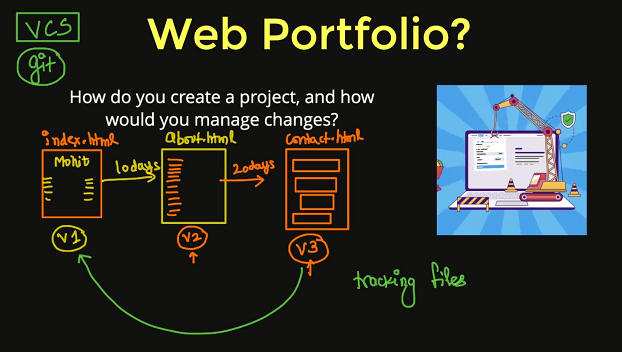
**Git & GitHub:**

* Let’s say we are developing a Code/Website/Software/Application?



**Challenge:**

1) How do you create a project & how would you manage changes?

**Solution:**

Version Control System i.e. VCS

**What is Version Control System?**

* Version Control System is a management software responsible for[managing changes to computer programs](https://en.wikipedia.org/wiki/Computer_program), documents, large websites, or other collections of information.
* Easy recovery of files/folders.
* Rollback to the previous versions.
* Inform us about Who, What, When, Why changes have been made.

**History/Evolution of VCS?**

**1) Local VCS:**

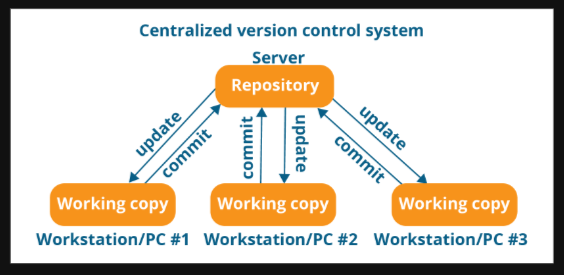
* Changes are stored in a database along with timestamp.
* Code is in Local System.

**Cons:**

* Project can be lost, if your hard-disk is corrupted.

**2) Centralized VCS:**

* **Repository:** A directory/folder where your project/code resides in a remote location on a server.
* Centralized VCS contain just one repository **i.e.** central repository and each user gets to work on the same BUT they do not have a local repository/folder.

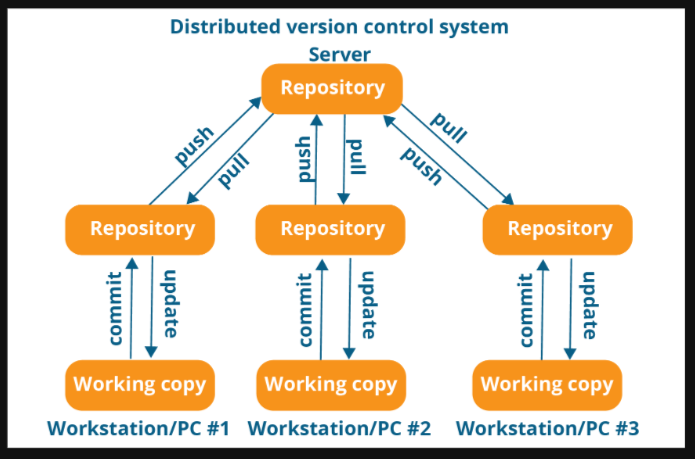


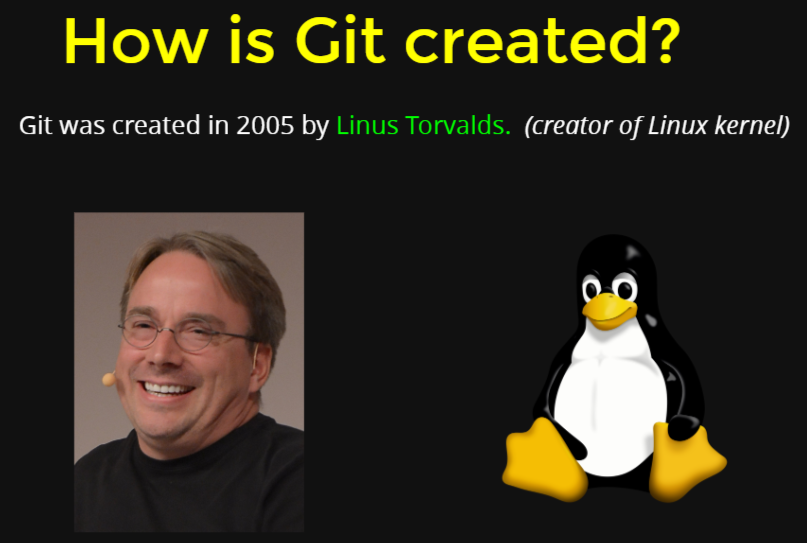
**Cons:**

* If the central repo goes down even for an hour.
* You'll lose the project if the hard disk of the central server goes down.

**Distributed VCS:**

* Distributed VCS contains multiple repositories. Each user has their own local repository & there is a central repository where the final code resides.
* Provides full Back-Up of the project.
* Git is an example of distributed VCS.





**Task:** Investigate the Evolution of Git?

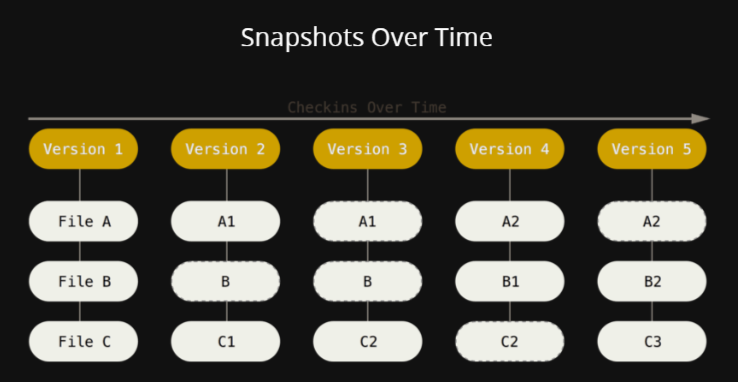
* BitKeeper Vs Linux Kernel

**What is Linux?**

Linux is an open source operating system that is made up of the kernel, the base component of the OS, and the tools, apps, and services bundled along with it.

**What is Git?**

1. **Git** is a free & open sourced **D**istributed **V**ersion **C**ontrol **S**ystem.
2. **Git** helps developers **keep** **track** of the **history** of their **code** **files** by **storing** them in **different** **versions** on its **own** **server** **repository**, i.e., **GitHub**.
3. Git stores **snapshot** of your project (not differences).



**Features of Git:**

* Collaboration
* Store Versions
* Analyse the code changes
* Distributed & Secure
* Almost Everything is Locally
* Branching

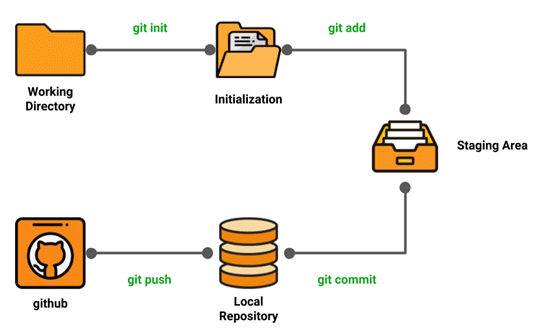
**What is GitHub?**

**GitHub** is a **cloud**-**based** hosting service that help developers **store** and **manage** their **code** as well as **track** and **control** **changes** to their **codes in** Git repositories.

**Difference between Git & GitHub:**

|  |  |
| --- | --- |
| **Git** | **GitHub** |
| Git is a tool to execute version control. | GitHub is a platform that utilizes Git to carry out its functionalities. |
| It is a software and in order to be used it has to be installed. | It is web based hosting service so there is no hassle of installation. |
| It is a command line tool. | It is a graphical user interface. |

**Git Life Cycle:**

****

**1) Local Working Directory**: The **first** **stage** of a **Git** **project** **life** **cycle** is the **local** **working** **directory** where our **project** resides, which **may** or **may** **not** be **tracked**.

**2) Initialization:** To **initialize** a **repository**, we give the **command** **git init.** With this command, we will make **Git** **aware** of the **project** **file** in our **repository**.

**3)** **Staging area:** Now that our **source** **code** **files**, **data** **files** and **configuration** **files** are being **tracked** by **Git** we will **add** the **files** that we **want** to **commit** to the **staging** **area** by the **git** **add** command. This process can also be called **indexing**.

**4) Commit:** Now, we will **commit** our **files** using the **git** **commit** -**m** ‘our message’ command.

We have **successfully** **committed** our **files** to the **local** **repository** but we **need** to **share** **projects** **files** with our **team** **members**.

**5)** This is when the **next** **stage** of the **Git** **life** **cycle** **occurs**, i.e. in **GitHub**, we **publish** our **files** from the **local** **repository** to the **remote** **repository**. We do that by using the **git** **push** command.

**Installation & Configuration of Git:**

git --version

git config --global user.name “<>”

git config --global user.email “<>”

git config --list

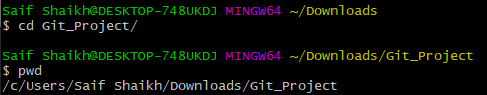
**Print the working Directory:**



**Create a Project Directory:**

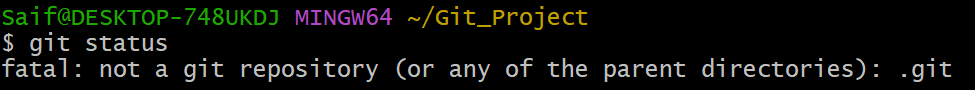


**Go to Project Folder & Print the working Directory:**



**1) git status:** This command will show the **modified** **status** of an **existing** **file** and the **file** **addition** **status** [**Tracked/Untracked**] of a **new** **file** if any that have to be staged/committed.

**Syntax: git status**

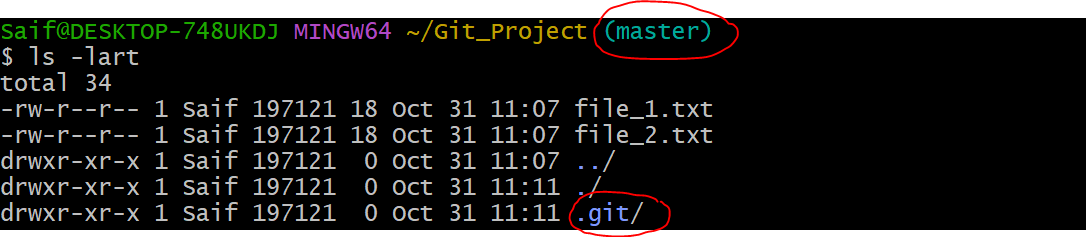


**2) git init:**

1. We have to **navigate** to our **project** **directory** and type the command **git init** to **initialize** a **Git** **repository** for our **local** **project** **folder**.
2. Git will create a hidden **.git** **directory/folder** and use it for **keeping** its **files** **organized** in **other** **subdirectories**.

**Syntax: git init**





**Help:**

git help 🡪 Help on CLI

git help git 🡪 Help on Web UI

git status -help 🡪 CLI Page

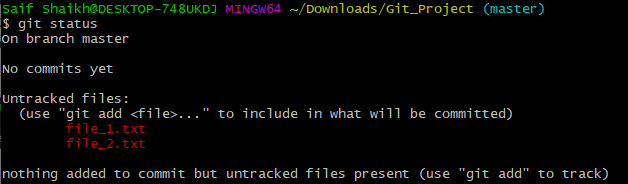
git help status 🡪 Web Page

**Create couple of files with some data:**

****

****

****



**3) git add:** This will **add** the **specified file(s)** into the **Git** **repository** **staging** area, where they are being **tracked** by **Git** and now **ready** to be **committed**.

**Syntax:**

**git add file(s)**

**git add . OR git add \* OR git add -A**

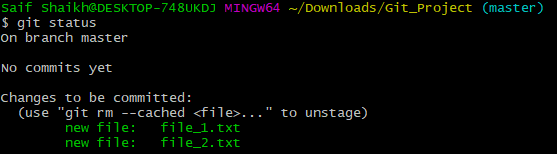
**Note:** By specifying **.** or **\*** it will **add** **all** the **files** **present** in the **current** **directory**.



**Note:**

1. Windows follows the original convention of a carriage return plus a line feed (CRLF) for line endings (\r\n)
2. Operating Systems like Linux and Mac use only the line feed (LF) character (\n)

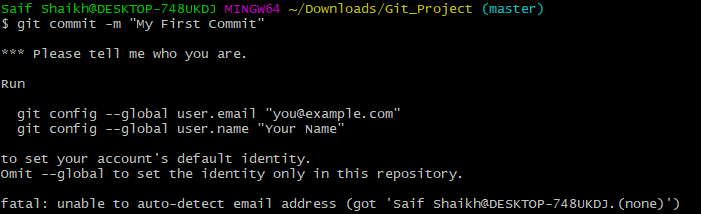
**Check the Status:**



1. **git commit:**
2. This command **records** **files** **permanently** in the **version** **history**.
3. All the **files**, which are there in the **directory** **right** **now** are being saved in the **Git** **file** **system**.

**Syntax: git commit -m “message”**

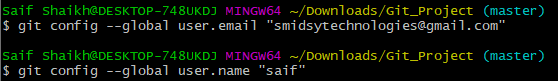
**Error:**



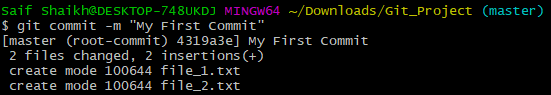
**Create a GitHub account & fire below command:**

git config --global user.email "smidsytechnologies@gmail.com"

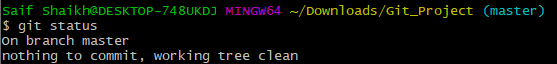
git config --global user.name "saif"



**Commit:**



**Check the status:**

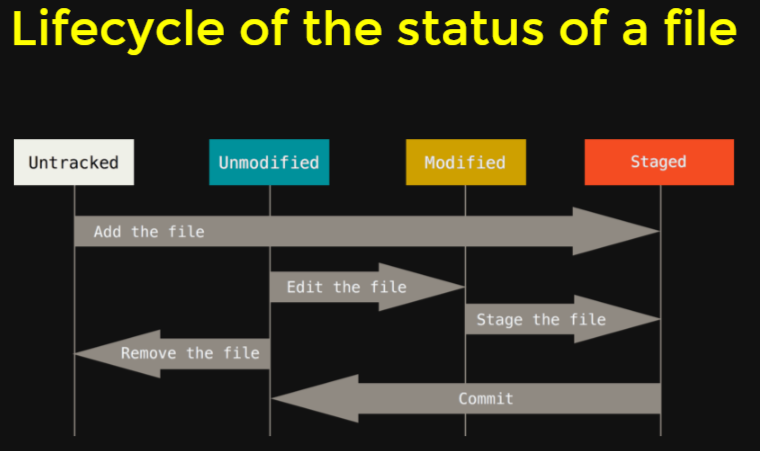


Tracking your files:

Each file in your working directory can be in one of two states: **tracked** or **untracked**.

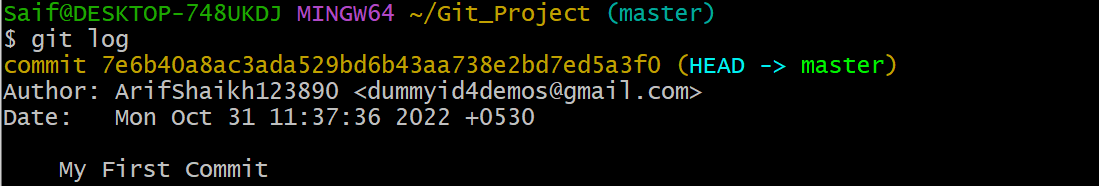
- **Tracked files** are files that Git knows about.

-  Any file in your working directory that was not in your last snapshot and is not in your staging area is **Untracked file**



**5) git log:** This command will show you all logs for the file requested.

**Syntax:** git log <filename>



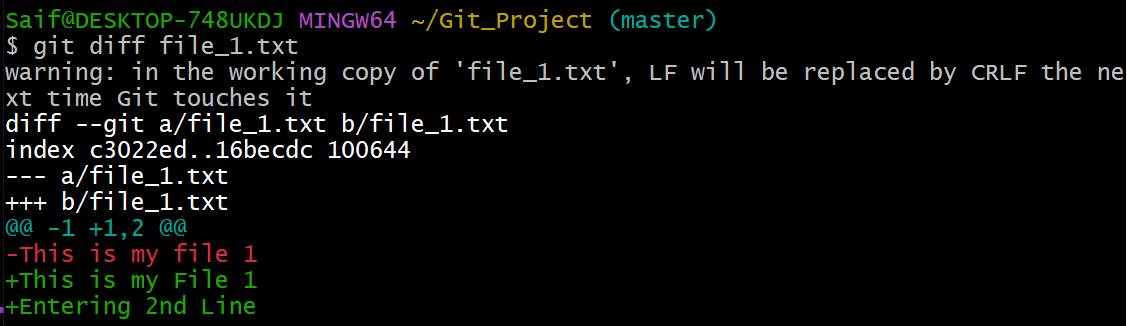
**git log --oneline**

**6) Skipping the staging area:**

**Syntax:** git commit -a -m "Commit Message"

**Note:** It cannot be done to the new files. There has to be one commit & then u can bypass staging area.

**7) git diff:** If you want to know exactly what you changed, not just which files were changed. Also if you have staged then you won’t see any difference.



**8) git ignore:**

1. Often times, we do not want git to keep track of some specific auto generated files like logs, or build etc.
2. To avoid them, you can list files or a pattern for files in **.gitignore** file.

touch error.log

git status

**touch .gitignore**



Now, open .gitignore & add error.log file into it & save it, now you won’t see that files but it will be under .gitignore. You can add regex patterns & directories also in this .gitignore file.

**9) Moving & Removing Files:** These command stage the changes automatically. You don't have to stage explicitly.

git rm <File\_Name>

git rm –f <File\_Name>

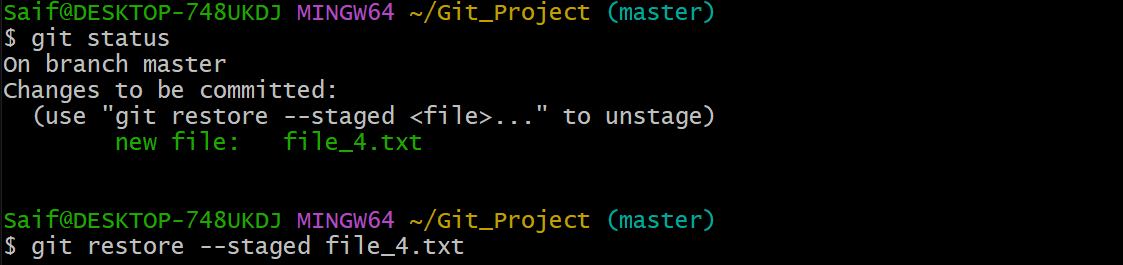
git mv <File\_Name> <New\_File\_Name>

**10) Unstaging Files:**

**git rm --cached** <File\_Name>

**git restore --staged** <File\_Name>

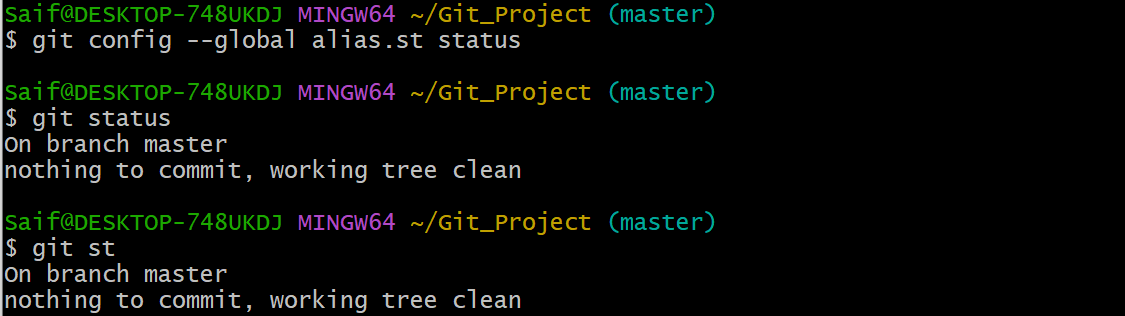
**Create file\_4.txt & Stage that file. After that check status & execute below command:**



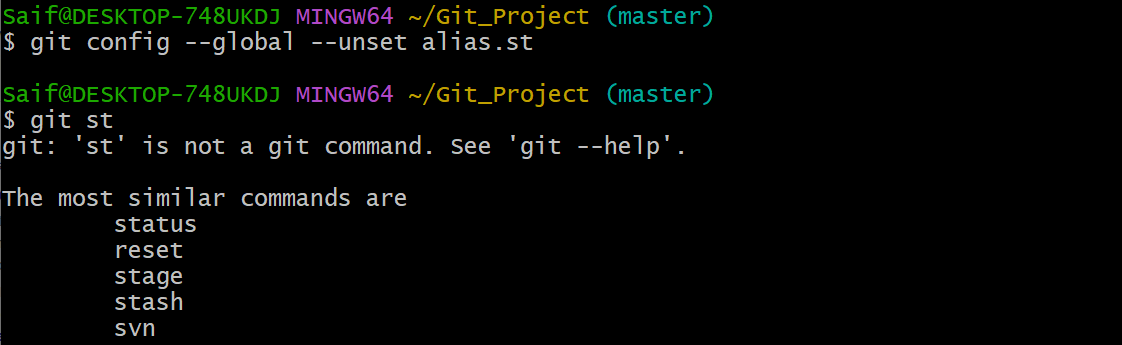
**12) Setting/Unsetting Alias:**

git config --global alias.st status

git st

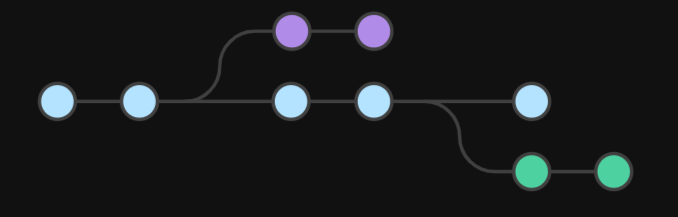


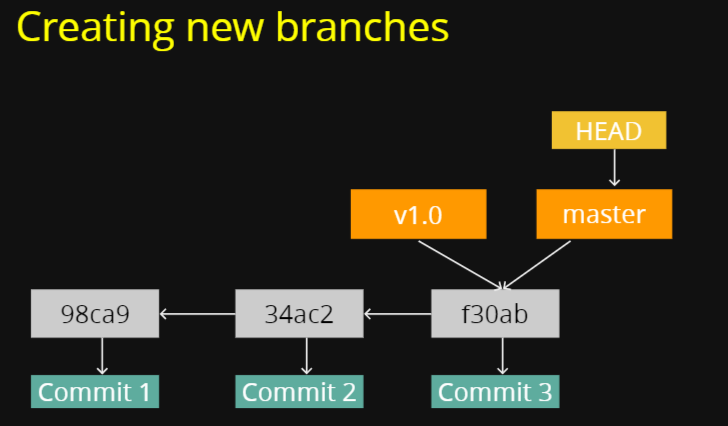
git config --global --unset alias.st



**What is branching?**

1. A branch in Git is simply a lightweight movable pointer to one of the commits.
2. The default branch name in Git is master
3. **Branching:** Diverging from the main branch.



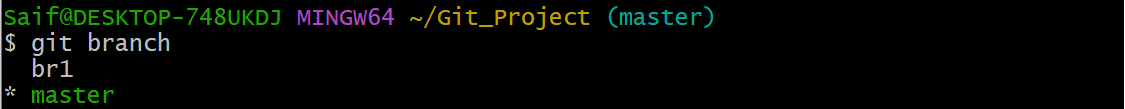


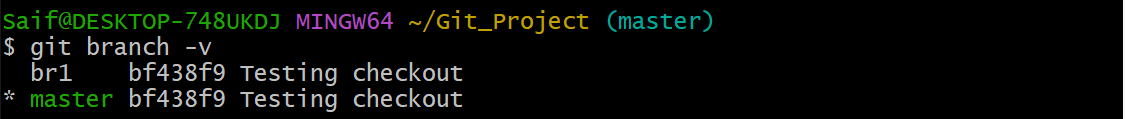
**13) Create a new Branch:**

git branch <Branch\_Name>

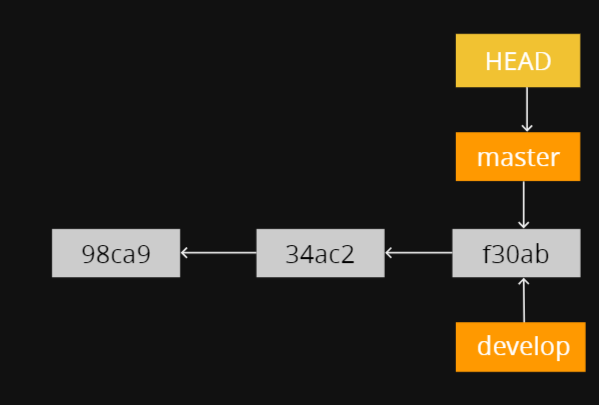
git branch br1

git branch





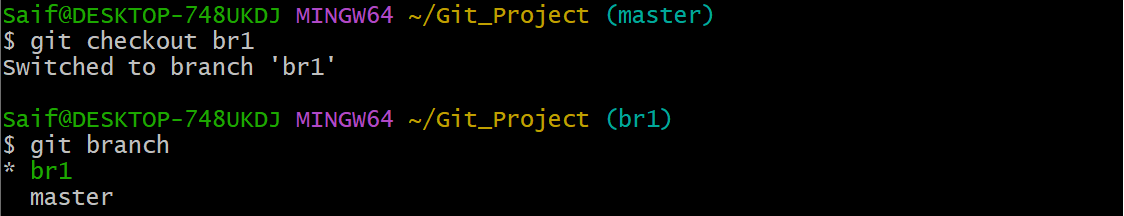
**Note:** Both master & branch will have same commit id & commit message.



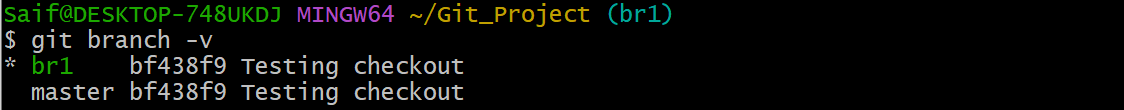
**14) Moving between branches:**

git checkout <Branch\_Name>

git checkout br1

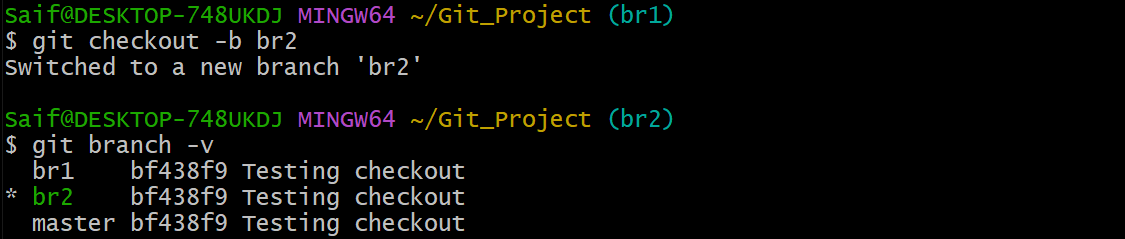


git branch –v

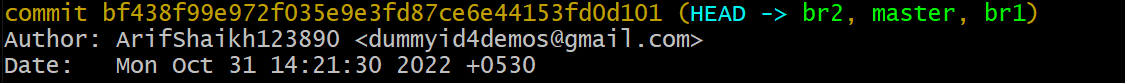


**Create New Branch & Switch to Newly Created Branch:**

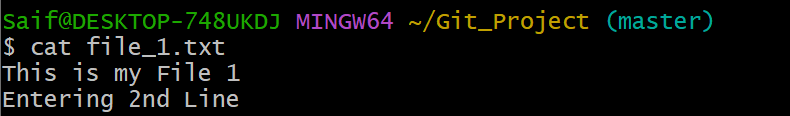
git checkout -b <Branch\_Name>



git log

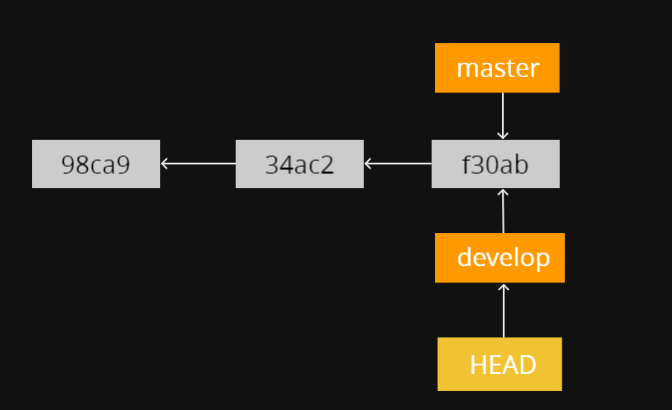


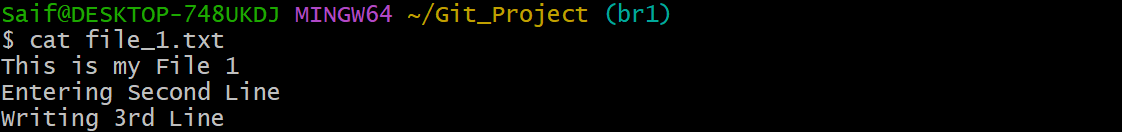
**Note:** Above the Head is pointing to br2.

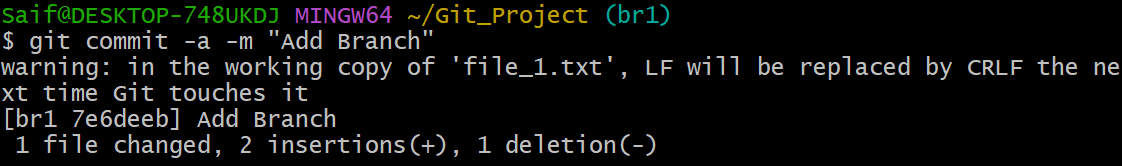


**Now, let’s checkout to br1:**

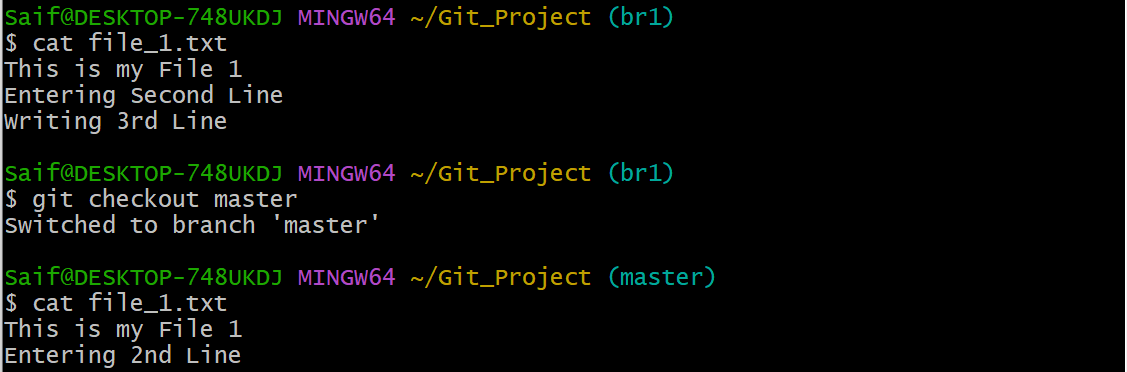
git checkout br1

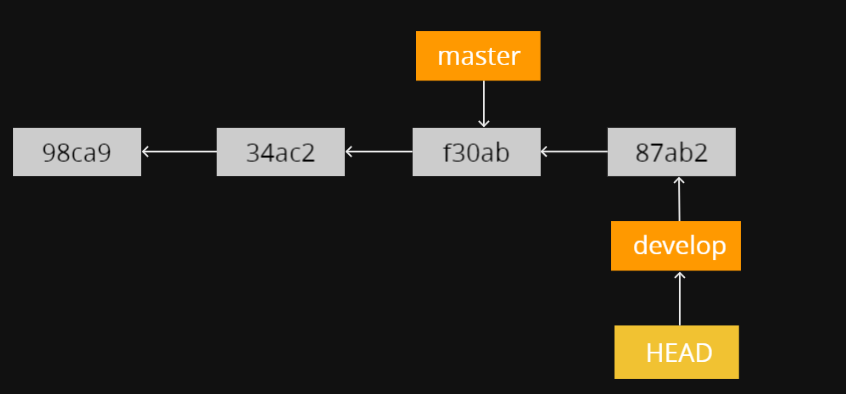




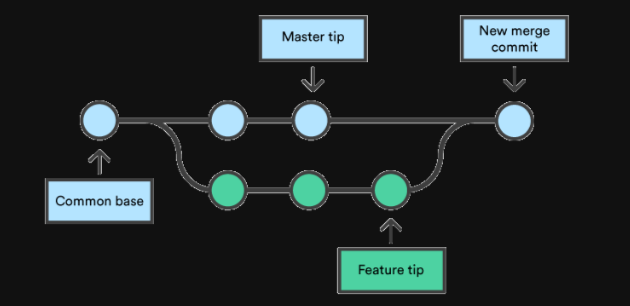


**Verify Data of Branch & Master:**



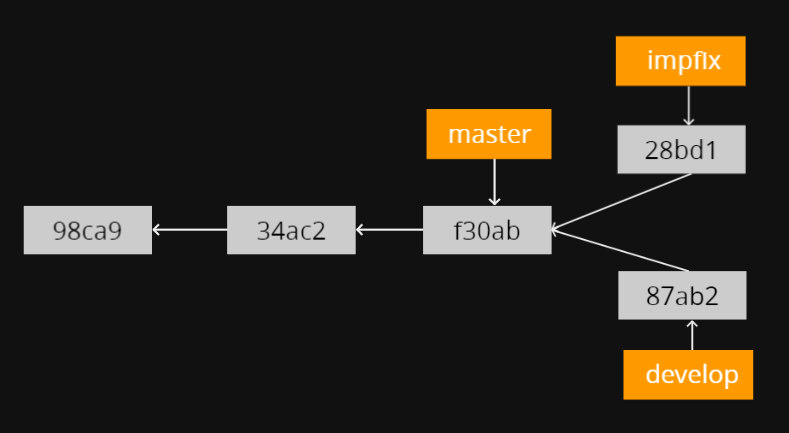


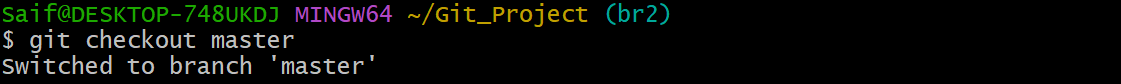
**15) How to Merge Branches?**

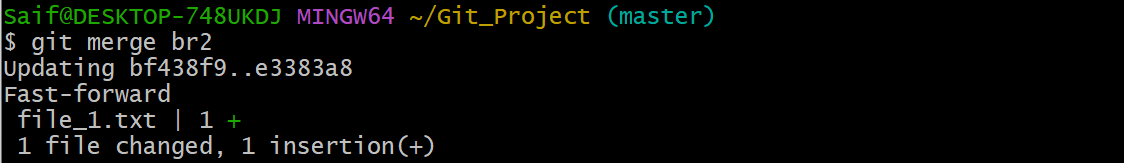


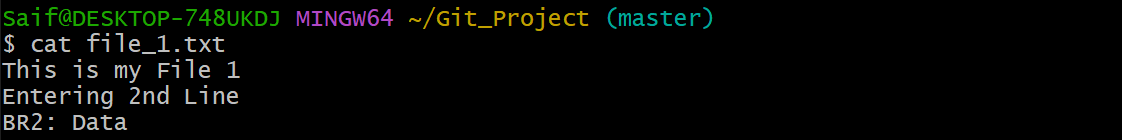
git checkout -b br2

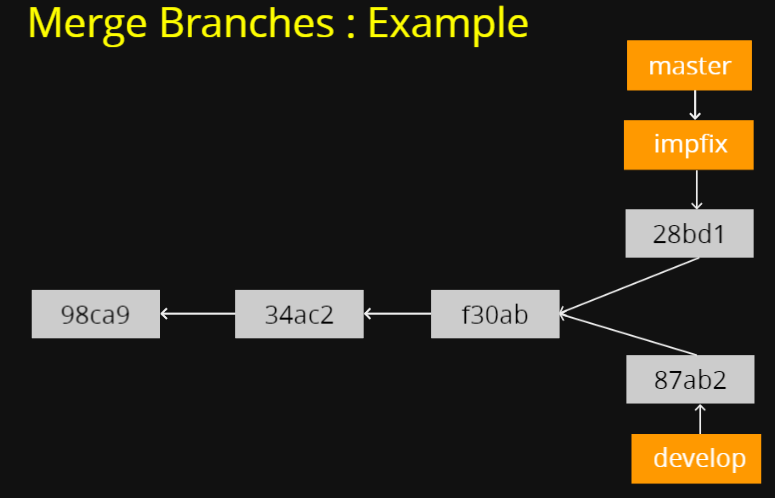
**Add some data to file\_1.txt, stage & commit.**

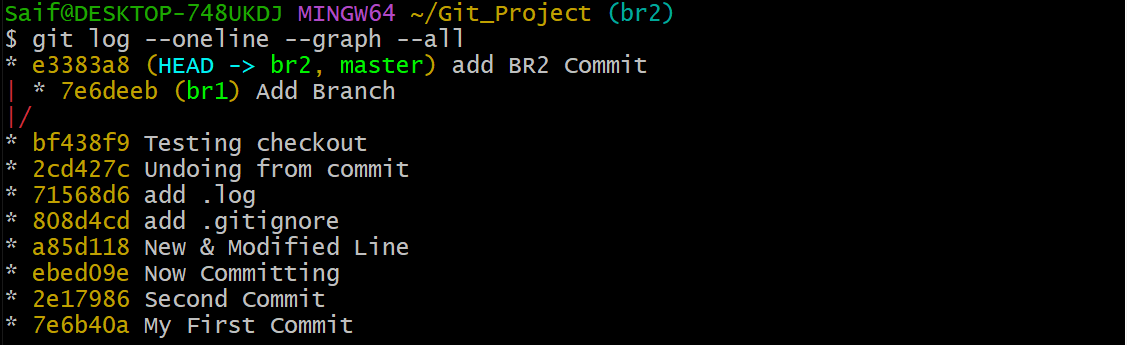








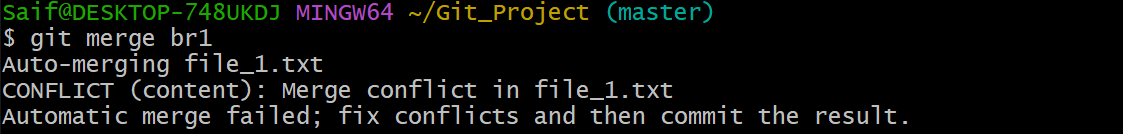


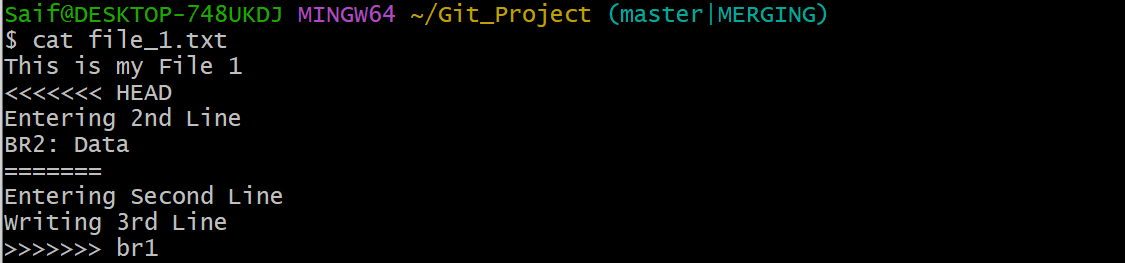


**Repeat the same process for br1 by checkout to master:**

a) Success

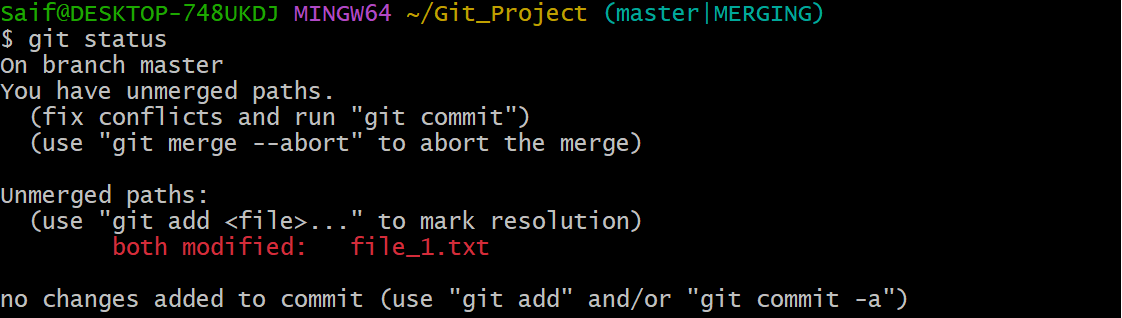
b) Conflict

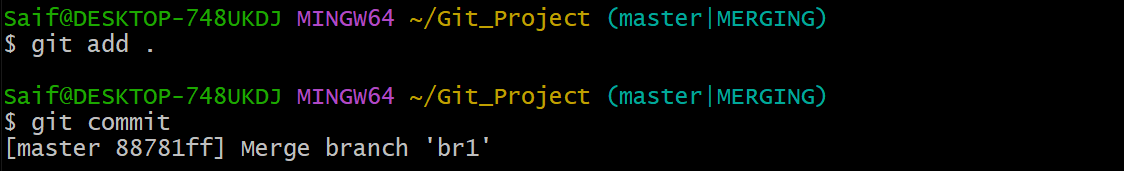




**Merge Conflicts:**

Changes in a file made by 2 different branches, trying to get merge results in a Merge Conflicts.





**Perform Manual Merge:**

vi file\_1.txt

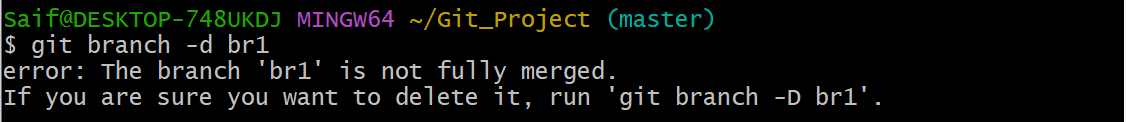
remove unnecessary codes/pointers

Save & Quite

**16) Delete Branch:**

**a) Deleting a Merged branch or No Changes branch:**

git branch -d <Branch\_Name>

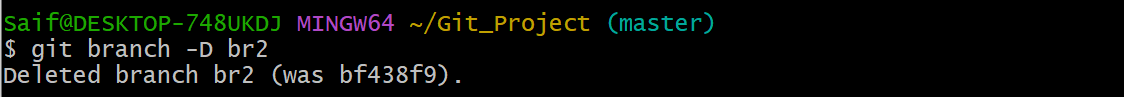


**Note:**

1. Before deleting a branch it needs to be merged.
2. But still if you want to delete without merging the you can use flag as -D

**b) Deleting a non-merged branch:**

git branch -D <Branch\_Name>

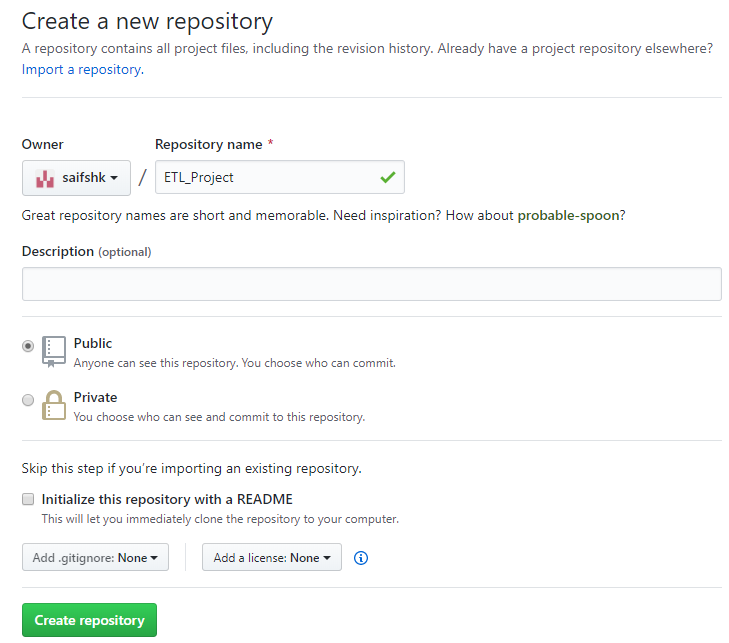


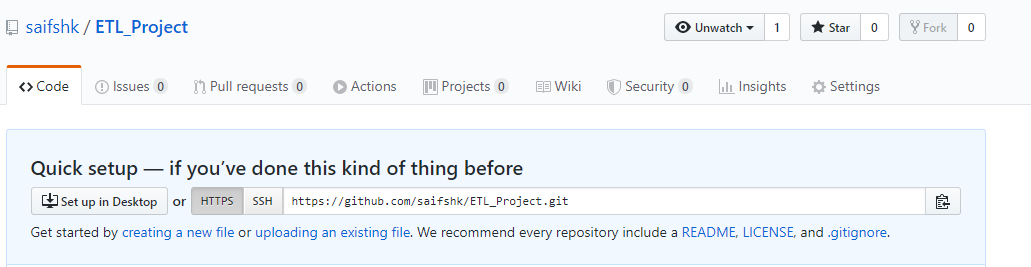
**Note:**

1. If there are no changes you can use flag as -d also to delete.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Now let’s push our final working code to GitHub**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Login to the **GitHub account** if the **account** **already** **exists** (If not, sign up on github.com). Now create a **repository** in **GitHub** where **you** **need** to **push** **your** **code**

'



**Click** on the **Copy** **icon** on the **right** **side** of the **URL** **box** of the **GitHub** **repository** to **copy** the **link**. Now, we are **ready** to **operate** the **remote** **commands** in our **repository** that we have **just** **created**.

git remote add origin <https://github.com/saifshk/ETL_Project.git>

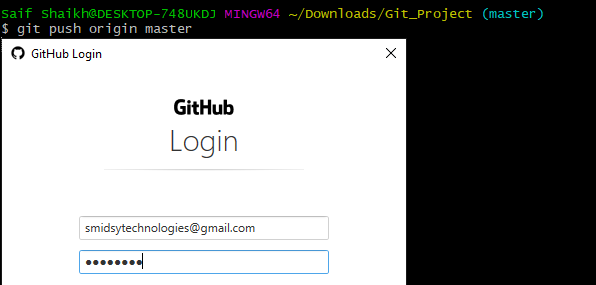


**17) git push:**

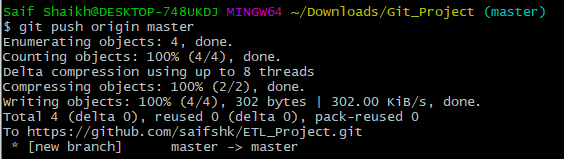
* Suppose, we have **made** **some** **changes** in the **file** and want to **push** the **changes** to our **remote** **repository** on a **particular** **branch**.
* By using the **command** ‘**git push**,’ the **local** **repository’s** **files** can be **synced** with the **remote** **repository** on **GitHub**.

**Syntax: git push origin [branch name]**

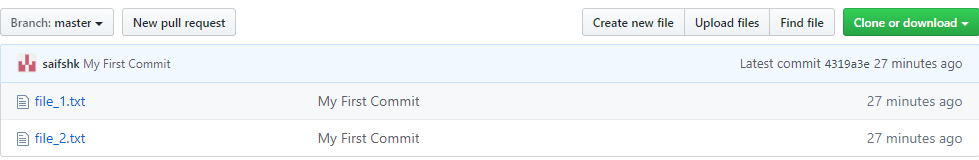
It will ask you **your** **username** & **password**:

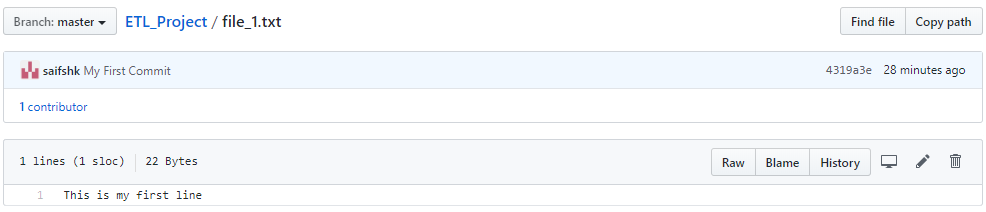


**Push Complete:**



**GitHub View to verify local repository is in sync with remote repository:**





**If you want to push to Main:**

git branch -M main

git status 🡪 Stage & Commit if any

git push origin main

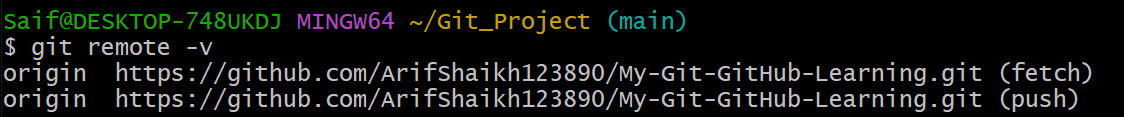
git push origin main

**Error:**

git pull origin main --allow-unrelated-histories

git push origin main

**Check Remote Repositories:**



**Pushing from one branch (Git) to other branch (GitHub)**

git push origin src\_branch**:**tgt\_branch

**Note:**

From **August 13, 2021**, **GitHub** is no longer accepting account passwords when authenticating Git operations. You need to add PAT (Personal Access Token)

**Generate PAT:**

ghp\_pZMKtofMSMnVXdmYnIkBvUo7K9nNZN2HyhSU 🡪 **Password**

git config --list **OR** git config --global --list 🡪 **Username**

**If your repository is in https then use PAT else ssh:**

**Windows:**

Control Panel 🡪 User Accounts 🡪 Windows Credentials 🡪 Find git:https://github.com 🡪 Edit 🡪 Replace Password of your GitHub PAT 🡪 Done

**Mac OS:**

Click on Spotlight Icon 🡪 Type Keychain Access & Press Enter 🡪 In Keychain Access search for github.com 🡪 Find Internet Password entry for github.com 🡪 Edit 🡪 Replace Password of your GitHub PAT 🡪 Done

**Reference Links:**

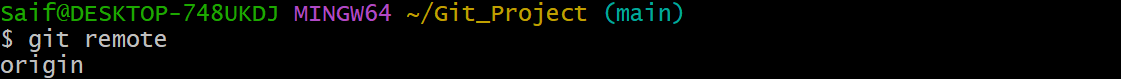
<https://github.blog/2020-12-15-token-authentication-requirements-for-git-operations/>

<https://stackoverflow.com/questions/68775869/message-support-for-password-authentication-was-removed-please-use-a-personal/68781050#68781050>

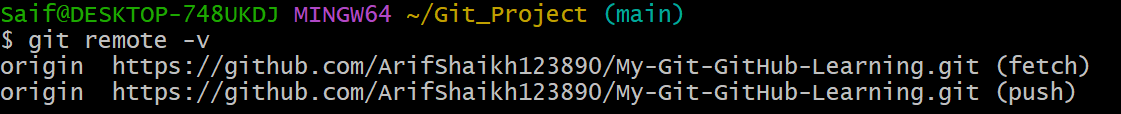
<https://docs.github.com/en/authentication/connecting-to-github-with-ssh>

**18) Understanding Remotes:**

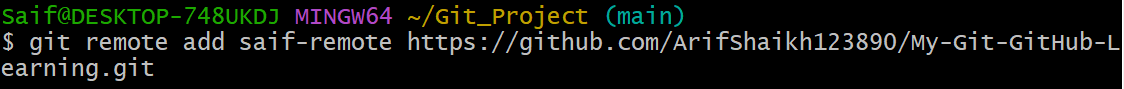
git remote

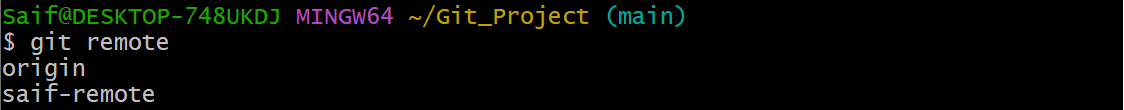


git remote -v



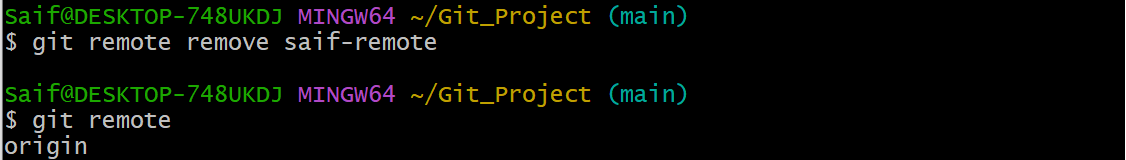
**Add a New Remote:**







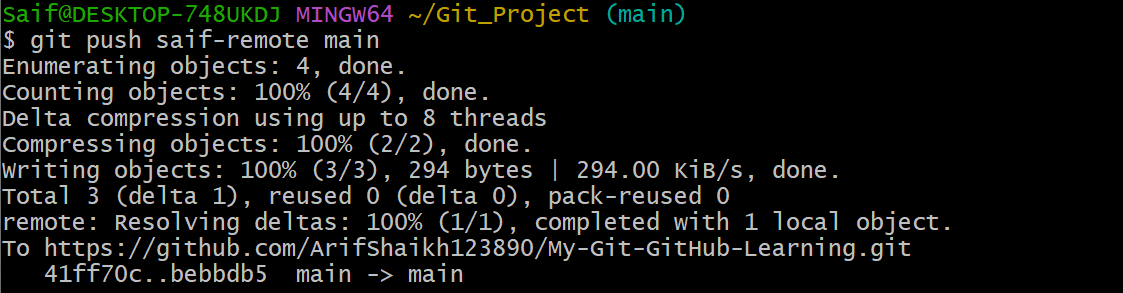
**Remove the Remote:**



**Add the Remote Again but paste the URL in notepad and then in GitBash:**

**Reference Link:**

[**https://stackoverflow.com/questions/53988638/git-fatal-protocol-https-is-not-supported**](https://stackoverflow.com/questions/53988638/git-fatal-protocol-https-is-not-supported)



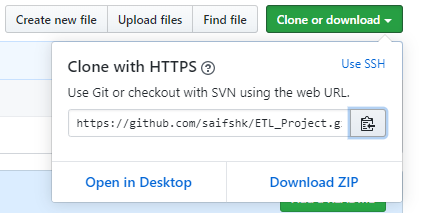
**19) git clone:** Suppose, we **want** to **work** on a **file** that is on a **remote** **GitHub** **repository** as another **developer**.

**Syntax: git clone [URL]**

**How can we do that?**

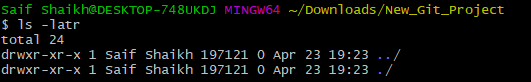
1. We can **work** on this **file** by **clicking** on **Clone or** **Download**and **copying** the **link** and **pasting** it on the **terminal** with the **git** **clone** **command**.
2. This will **import** the **files** of a **project** from the **remote** **repository** to our **local** **system**.

The below screenshot shows from where to copy the link:



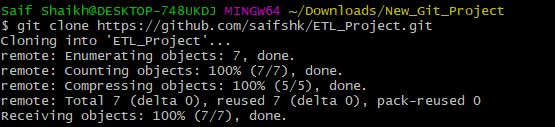
Now, **paste** the **copied** **link** along with the **git** **clone** command as shown below:

**Before Clone:**

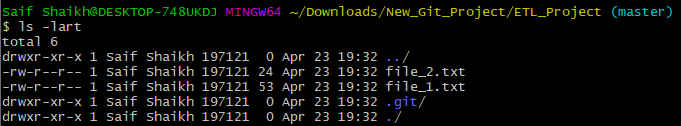


**Note**: Here, we **don’t** have to **use** the **git remote add origin** command because we have already **synced** the **remote** **repository** in the **local** **directory**. Now, if we **push** any **new** **file** it **knows** **where** it **has** to **go**.

**git clone https://github.com/saifshk/ETL\_Project.git**



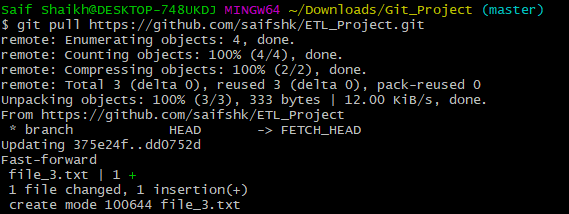
**After Clone:**



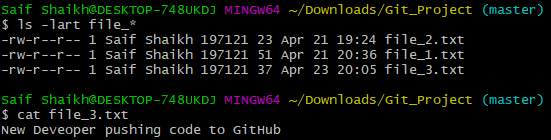
**20) git pull:** The **git** **pull** command **first** **runs** ‘**git** **fetch’** which **downloads** the **content** from the **specified** **remote** **repository** and then **immediately** **updates** the **local** **repo** to **match** the **content i.e. ‘git merge’**.

git pull = git fetch + git merge

**Syntax: git pull origin master OR git pull <url>**



**After pull:**

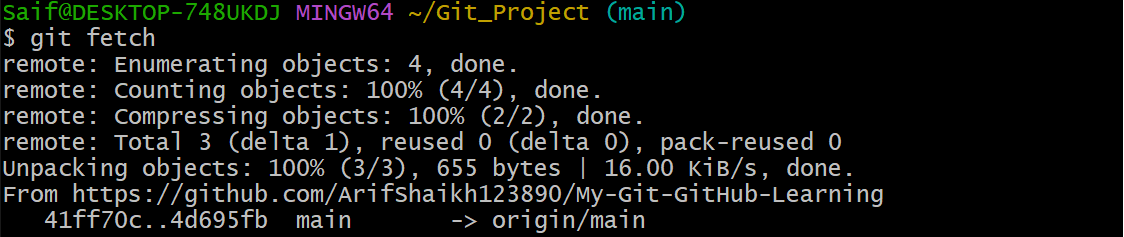


**21) git fetch:**

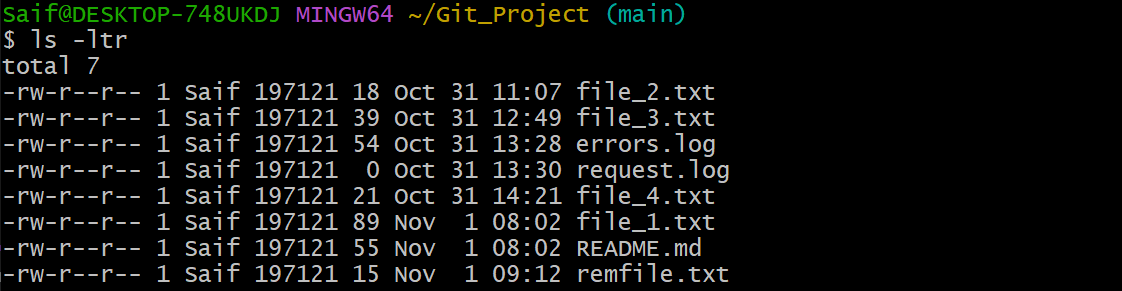
1. The git fetch command downloads commits, files and refs from a remote repository into your local repository.
2. It only fetches the changes from remote repository to local repository but the change is not reflected.
3. To display/reflect you need to execute one more command i.e. git merge

**Note:**

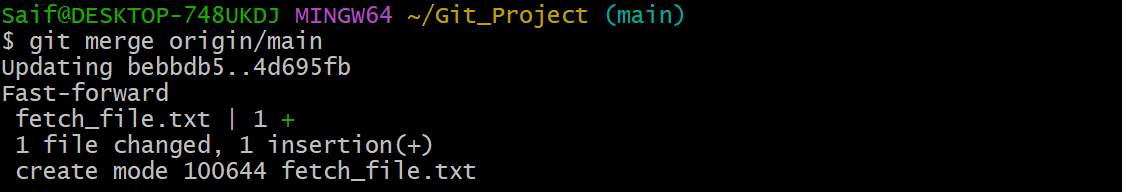
clone will always copy the entire repository which is not a good idea always hence we use fetch so that it absorbs only changes.

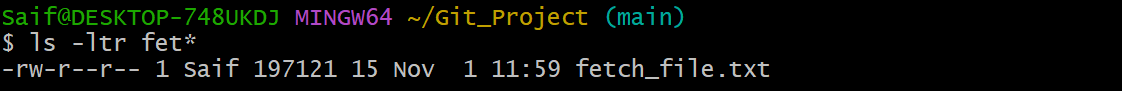


**But you do not see those file in working directory its only available in local repository.**



**To make it available in working directory you need to execute git merge**





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*