GIT AND GITHUB

GIT:

Git is a popular version control system software .

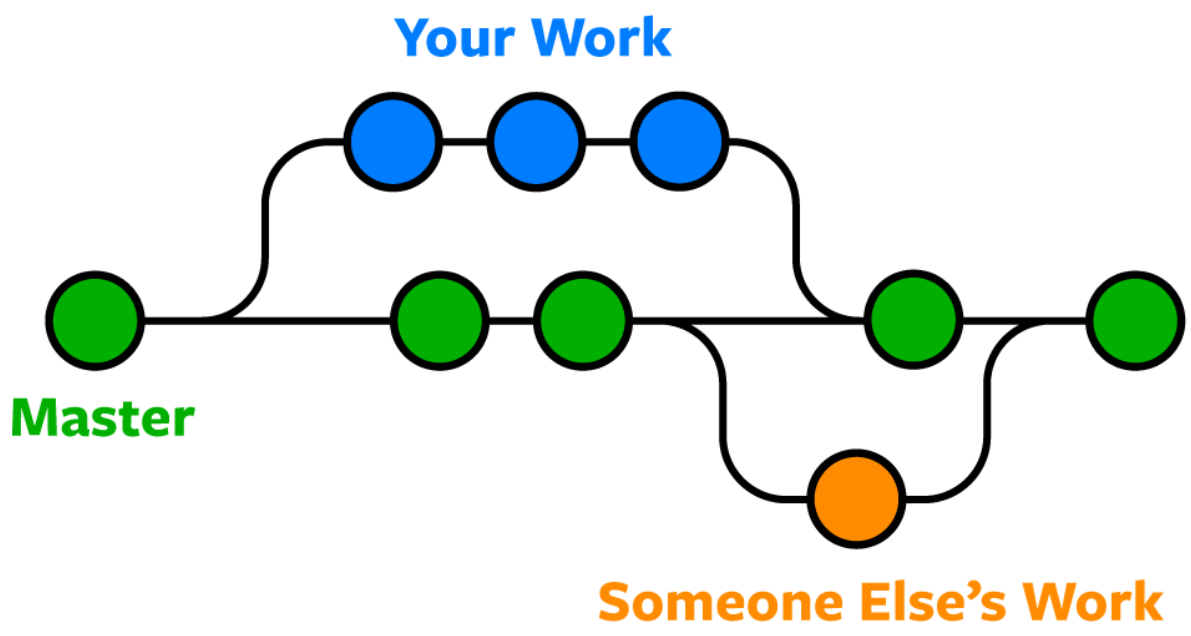
It was created by Linus Torvalds in 2005.

It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

Key Git Concepts

- **Repository:** A folder where Git tracks your project and its history.
- **Clone:** Make a copy of a remote repository on your computer.
- **Stage:** Tell Git which changes you want to save next.
- **Commit:** Save a snapshot of your staged changes.
- **Branch:** Work on different versions or features at the same time.



- **Merge:** Combine changes from different branches.
- **Pull:** Get the latest changes from a remote repository.
- **Push:** Send your changes to a remote repository.

GITHUB:

It is an online service hosting that GIT software.

VERSION CONTROL SYSTEM

A **Version Control System (VCS)** is a tool that helps track and manage changes to a project's codebase over time.

It allows **multiple developers** to work on the same project simultaneously without conflicts, maintains a history of all changes, and enables easy rollback to previous versions if needed. VCS ensures **collaboration**, **code integrity**, and efficient management of **software development**.

Types of Version Control Systems

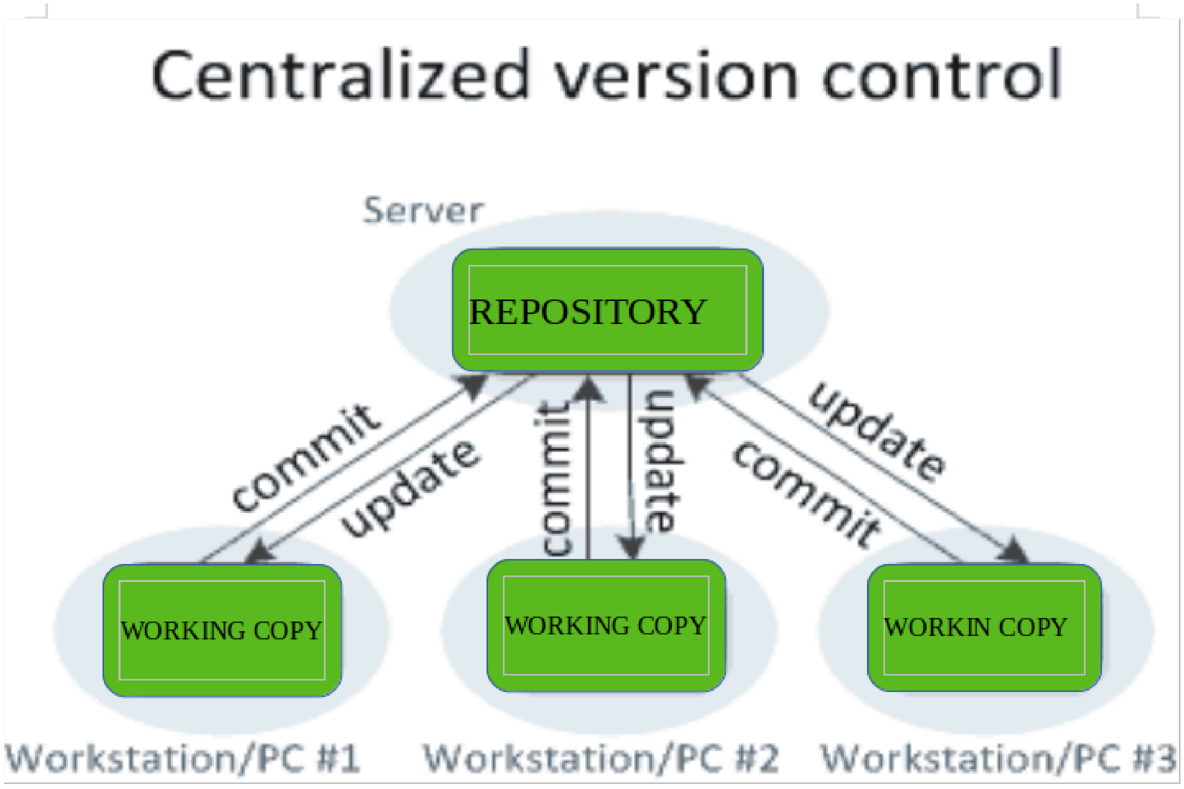
There are two main types of Version Control Systems: Centralized Version Control Systems (CVCS) and Distributed Version Control Systems (DVCS).

1. Centralized Version Control Systems

Centralized version control systems contain just one repository globally, and every user needs to commit for reflecting one's changes in the repository. It is possible for others to see your changes by updating.

Two things are required to make your changes visible to others

- You commit
- They update



- The **benefit** of CVCS (Centralized Version Control Systems) makes collaboration amongst developers along with providing an insight to a certain extent on what everyone else is doing on the project.
- It allows administrators to fine-grained control over who can do what. It has some **downsides** as well which led to the

development of DVS.

- The most obvious is the single point of failure that the centralized repository represents if it goes down during that period collaboration and saving versioned changes is not possible.

Advantages of CVCS

- Simplicity in setup and management.
- Easy to maintain a single central repository.
- Suitable for small teams or projects with limited collaboration needs.

Disadvantages of CVCS

- If the central server goes down, no one can commit or retrieve updates.
- Limited support for branching and merging compared to DVCS.
- It can become a bottleneck if many developers are committing at once.

Example:**Subversion (SVN),**

2. Distributed Version Control Systems

Distributed version control systems contain multiple repositories. Each user has their own repository and working copy. Just committing your changes will not give others access to your changes. This is because commit will reflect those changes in your local repository and you need to push them in order to make them visible on the central repository.

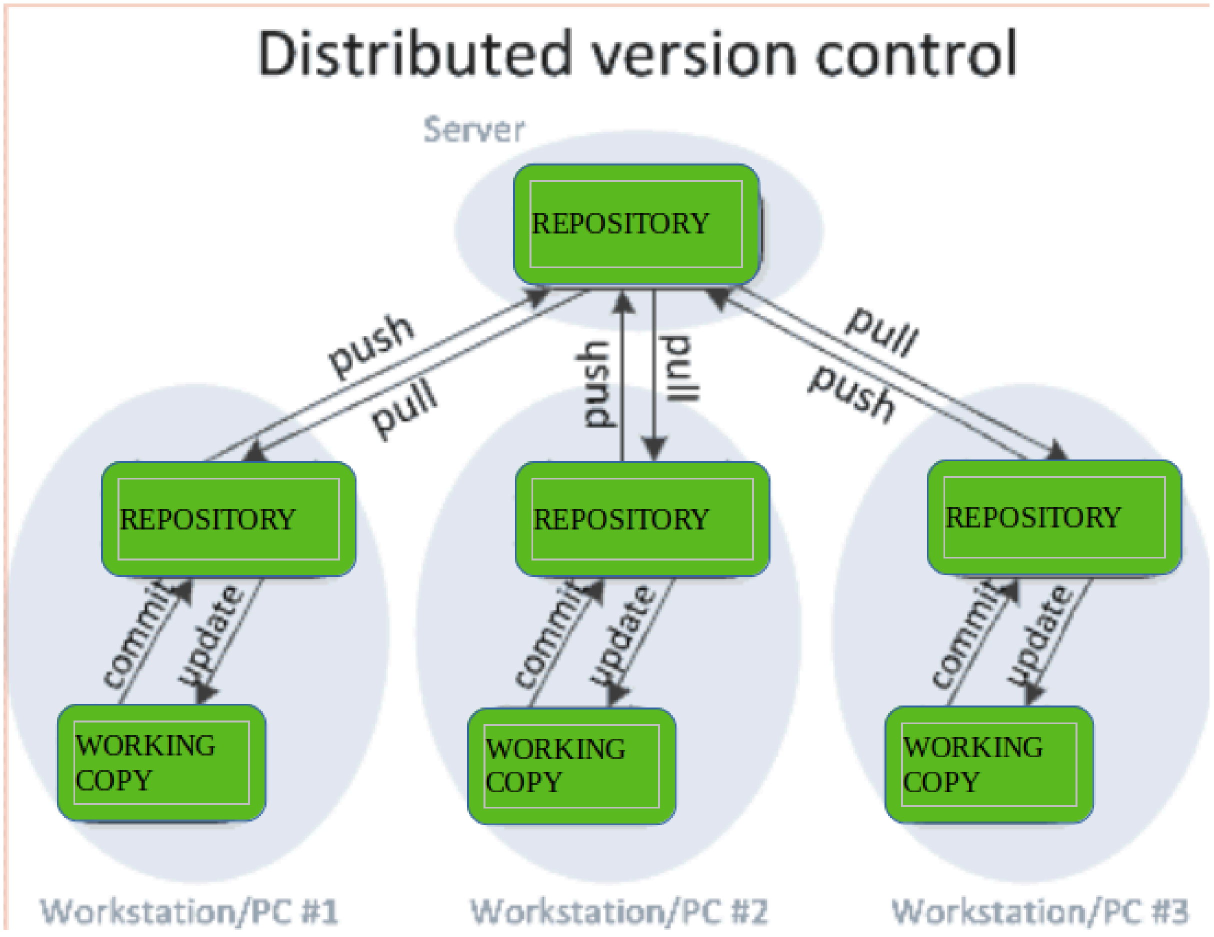
Similarly, When you update, you do not get others' changes unless you have first pulled those changes into your repository.

To make your changes visible to others, 4 things are required:

- You commit
- You push

- They pull
- They update

The most popular distributed version control systems are Git, and Mercurial. They help us overcome the problem of single point of failure.



Advantages of DVCS

- Allows developers to work offline and commit changes locally before syncing with others.
- Better handling of branches and merges.
- Faster access to version history, as developers don’t have to rely on a central server.
- More resilient; if one copy of the repository is lost, it can be recovered from others.

Disadvantages of DVCS

- More complex setup and configuration.
- Can require more storage space as each developer has a full repository.
- Potentially higher bandwidth usage when pushing and pulling from central servers.

GIT TERMINOLOGIES

1. Git Repository

A Git Repository is a directory that contains a git working stage. The working directory contains all the files that are tracked by Git. The Git repository is the place where the files are stored. We can think of a Git repository as a file system. Where the files are stored. Git Repository has a directory structure. The directory structure is similar to the directory structure of a file system.

(`git —version`) —> to view Git’s current version.

(`git —status`) —> to view status or repositries.This is used to view whether a repo or file has been saved with the changes to the staging area or not.

Use `git config` command in your terminal to manage Git-related settings on your computer:

- To list **all** properties: `git config --list`
- To list **system** properties: `git config --list --system`
- To list **global** properties: `git config --list --global`
- To list **local** (repository) properties: `git config --list --local`

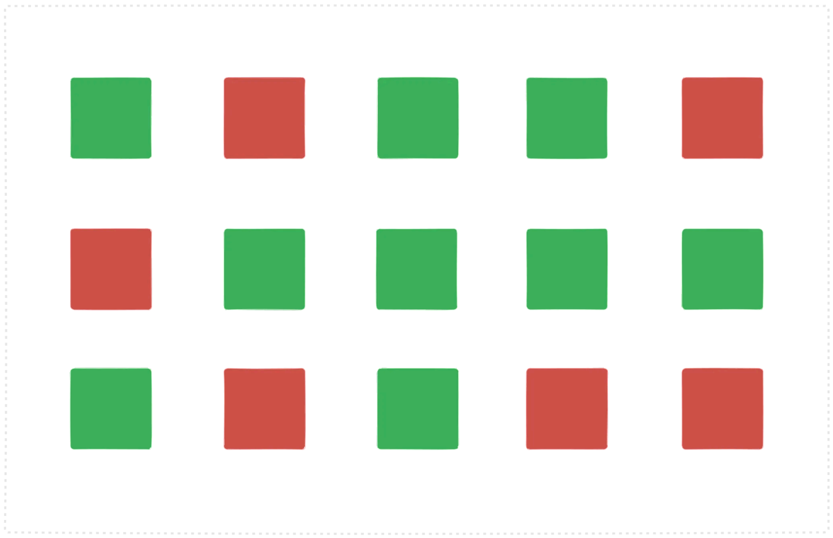
- To set a **system** property: `git config --system <someproperty> <value>`
- To set a **global** property: `git config --global <someproperty> <value>`
- To set a **local** (repository) property: `git config --local <someproperty> <value>`, or you can skip the `--Local` flag

Additionally, if you want to know an explicit config location and scope, use the following:

- To show where the setting comes from: `git config --list --show-origin`
- To show to which scope the particular setting belongs: `git config --list --show-scope`
- To change whole config file : `git config --global  --edit`

Creating Repository

- `git status`:   to view if the repo is already initialized or not.
- `git init :` this initialize that directory as repo. Starting tracking a folder.



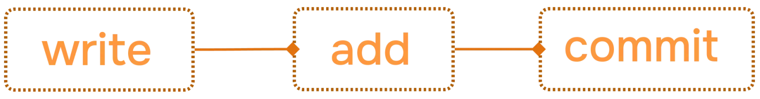
🟩 Tracked 🟥 Not Tracked

Not all folders are meant to be tracked by git. Here we can see that all green folders are projects are getting tracked by git but red ones are not.

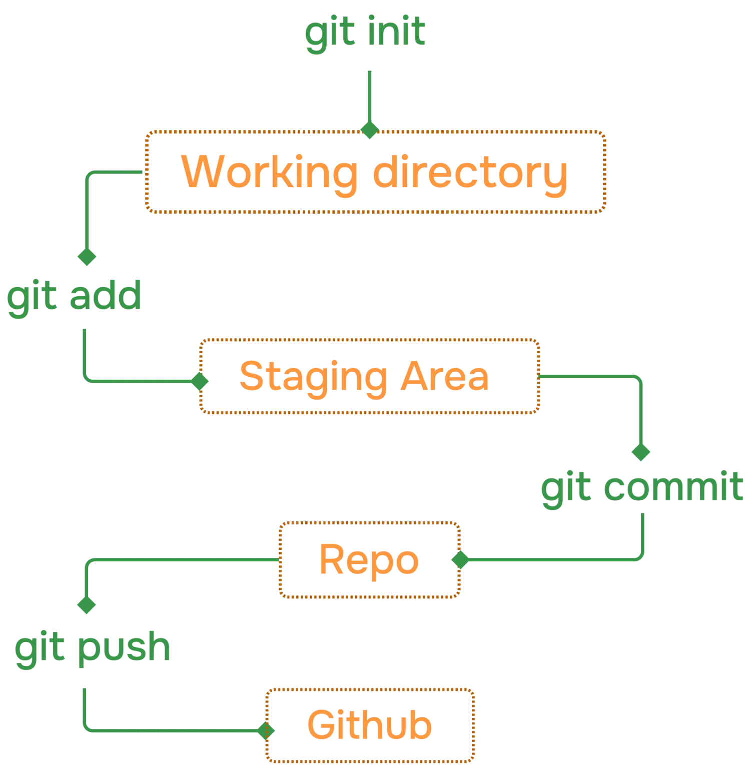
Commit

commit is a way to save your changes to your repository. It is a way to record your changes and make them permanent. You can think of a commit as a snapshot of your code at a particular point in time. When you commit your changes, you are telling git to save them in a permanent way. This way, you can always go back to that point in time and see what you changed.

Usual flow looks like this:



GIT WORKFLOW



When you want to track a new folder, you first use `init` command to create a new repository. Then you can use `add` command to add the folder to the repository. After that you can use `commit` command to save the changes. Finally you can use `push` command to push the changes to github . Of course there is more to it but this is the basic flow.

GIT ADD:

Taking data to the staging area. Stage is a way to tell git to track a particular file or folder. You can use the following command to stage a file:

`git add <file1> <file2>` ⇒ take file1 and file2 to staging area!

`git add .` ⇒ to add all the files in pwd in staging!

`git status`

Here we are initializing the repository and adding a file to the repository. Then we can see that the file is now being tracked by git. Currently our files are in staging area, this means that we have not yet committed the changes but are ready to be committed.

`use "git rm --cached <file>..." to unstage`

How This works?

`$sha1=$(git hash-object -w example.file)`

`$$sha1`

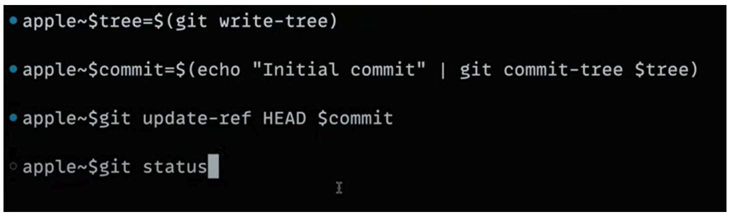
`git update-index --add --cacheinfo 100644 $sha1 example.file`

Commit

`git commit -m "commit message"`

`git status`

Here we are committing the changes to the repository. We can see that the changes are now committed to the repository. The `-m` flag is used to add a message to the commit. This message is a short description of the changes that were made. You can use this message to remember what the changes were. Missing the `-m` flag will result in an action that opens your default settings editor, which is usually VIM.

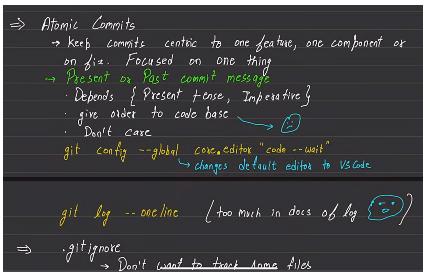


```
apple~$tree=$(git write-tree)

apple~$commit=$(echo "Initial commit" | git commit-tree $tree)

apple~$git update-ref HEAD $commit

apple~$git status
```



⇒ Atomic Commits
→ keep commits centric to one feature, one component or on fix. Focused on one thing
→ Present or Past commit message
· Depends { Present tense, Imperative }
· give order to code base
· Don't care
git config --global core.editor "code --wait"
↪ changes default editor to VS Code

git log --oneline (too much in docs of log)

⇒ .gitignore
→ Don't want to track some files

Logs

`git log`

This command will show you the history of your repository. It will show you all the commits that were made to the repository. You can use the `--oneline` flag to show only the commit message with a unique id of that commit. This will make the output more compact and easier to read.

> Atomic commits are a way to make sure that each commit is a self-contained unit of work. This means that if one commit fails, you can always go back to a previous commit and fix the issue. This is important for maintaining a clean and organized history in your repository.

gitignore

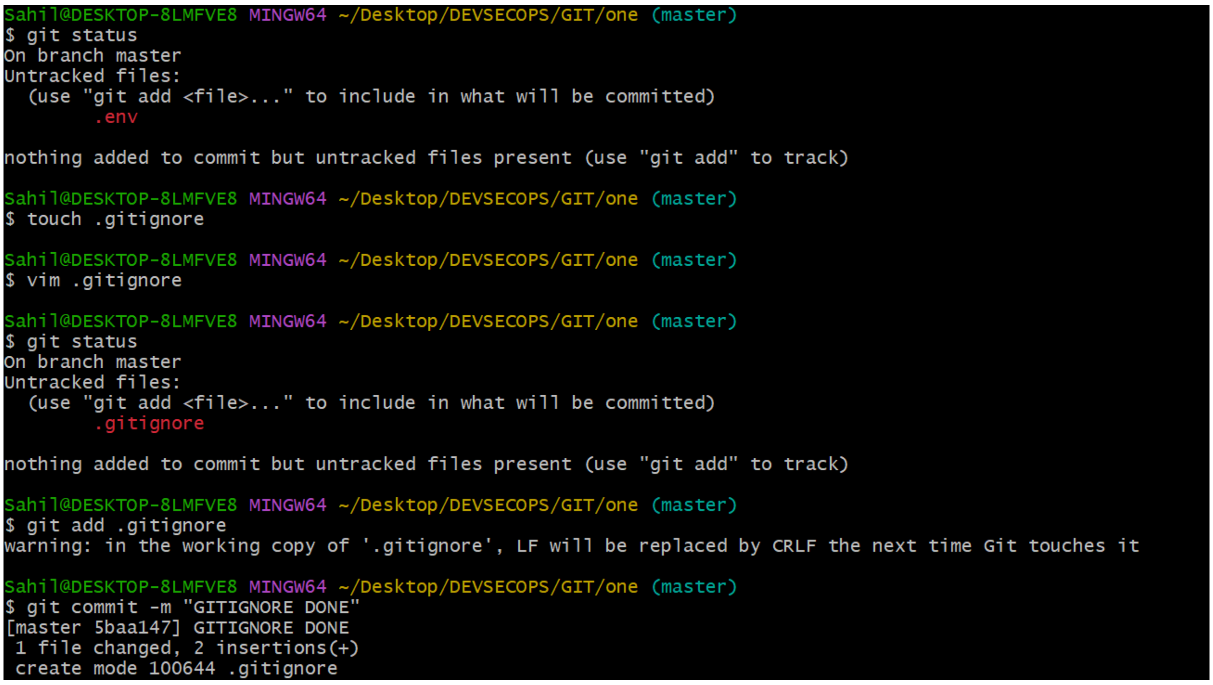
Gitignore is a file that tells git which files and folders to ignore. It is a way to prevent git from tracking certain files or folders. You can create a gitignore file and add list of files and folders to ignore by using the following command:

Example:

.gitignore

`node_modules.env.vscode`

Now, when you run the `git status` command, it will not show the `node_modules` and `.vscode` folders as being tracked by git.



```
Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .env

nothing added to commit but untracked files present (use "git add" to track)

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ touch .gitignore

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ vim .gitignore

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git add .gitignore
warning: in the working copy of '.gitignore', LF will be replaced by CRLF the next time Git touches it

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git commit -m "GITIGNORE DONE"
[master 5baa147] GITIGNORE DONE
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

.GitKeep

As by default git doesn’t allow to take empty folders to staged to repo form but by using the .gitkeep in the empty folder we can keep that folder even if it doesn’t have anything!

Git Snapshots

A git snapshot is a point in time in the history of your code. It represents a specific version of your code, including all the files and folders that were present at that time. Each snapshot is identified by a unique hash code, which is a string of characters that represents the contents of the snapshot.

A snapshot is not an image, it’s just a representation of the code at a specific point in time. Snapshot is a loose term that is used when git stores information about the code in a locally stored key-value based database. Everything is stored as an object and each object is identified by a unique hash code.

3 Pillars of Git

The three Pillar of git are:

- Commit Object
- Tree Object
- Blob Object

Commit Object

Each commit in the project is stored in `.git` folder in the form of a commit object. A commit object contains the following information:

- Tree Object
- Parent Commit Object
- Author
- Committer
- Commit Message

Tree Object

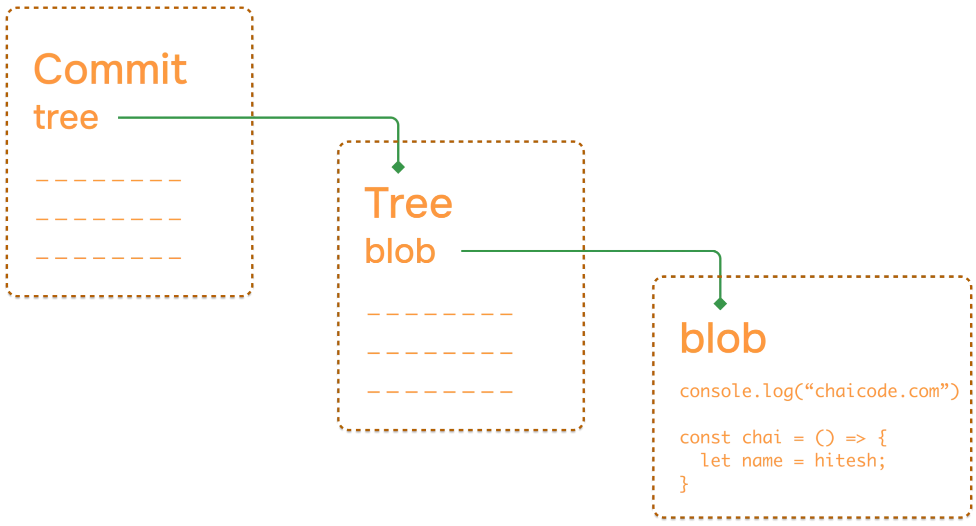
Tree Object is a container for all the files and folders in the project. It contains the following information:

- File Mode
- File Name
- File Hash
- Parent Tree Object

Everything is stored as key-value pairs in the tree object. The key is the file name and the value is the file hash.

Blob Object

Blob Object is present in the tree object and contains the actual file content. This is the place where the file content is stored.



Helpful commands

`git show -s --pretty=raw <commit-hash>`

Grab tree id from the above command and use it in the following command to get the tree object:

`git ls-tree <tree-id>`

Grab th blob id from the above command and use it in the following command to get the blob object:

`git show <blob-id>`

Grab tree id from the above command and use it in the following command to get the commit object:

`git cat-file -p <commit-id>`

Git Branches

In Git, a `branch` is like a separate workspace where you can make changes and try new ideas without affecting the main project. Think of it as a "parallel universe" for your code. Branches let you work on different parts of a project, like new features or bug fixes, without interfering with the main branch. The main project is usually called the main or master branch. When you're done with your work on the branch, you can merge your changes back into the main project. Git branches help you manage different tasks at the same time, making it easier to work together with others without messing up the main code.

Why Use Git Branching?

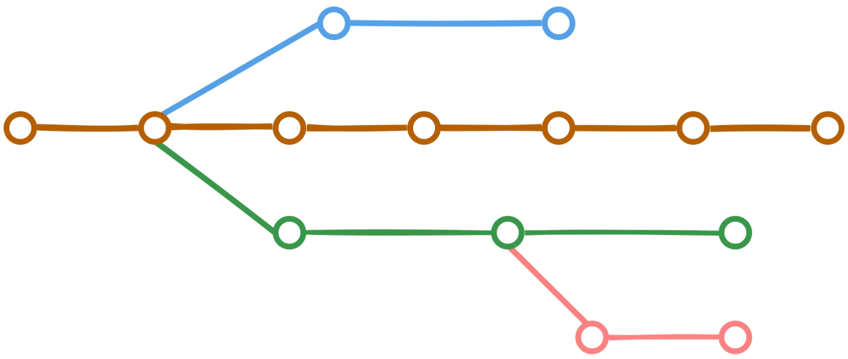
- **Isolated Development**: You can work on different features or bug fixes in isolation without affecting the main project or other team members’ work.
- **Parallel Development**: Multiple developers or teams can work on different tasks at the same time without interfering with each other.
- **Safe Experimentation**: Branches allow you to experiment with new features or changes without worrying about breaking the main codebase.
- **Efficient Collaboration**: Multiple people can work on different parts of the project and later merge their work without conflicts.

How Git Branches Work?

Git branches are like separate paths where you can work on different tasks without disturbing the main project.

- **Main Branch**: This is the main path where your working code (the one that's usually stable) is stored.
- **Feature Branch**: When you want to add something new, like a feature or bug fix, you create a new branch. It’s like a copy of the main path, where you can work freely.
- **Commit**: As you work on your feature or bug fix, you make changes (called "commits") on your feature branch, keeping it separate from the main branch.
- **Merge**: Once you're happy with your work, you combine (merge) the feature branch back into the main branch.

`git branch` : to see current branch.



HEAD in git

The HEAD is a pointer to the current branch that you are working on. It points to the latest commit in the current branch. When you create a new branch, it is automatically set as the HEAD of that branch.

> the default branch used to be master, but it is now called main. There is nothing special about main, it is just a convention.
>
> `git branch:` For seeing current branch and available branches
>
> `git branch bug-fix:` for creating branch followed by name{bug-fix}.
>
> `git switch bug-fix:` for switching from current branch to that another branch.
>
> `git log:` see the logs succesfully commited.
>
> `git switch main:` This command switches to the `main` branch.{observer those file we created

> in bug-fix are not available her}.
>
> `git switch -c dark-mode: ` - This command creates a new branch and switch to it called `dark-mode`. the `-c` flag is used to create a new branch.
>
> `git checkout orange-mode:` This command switches to the `orange-mode` branch.

GIT MERGE

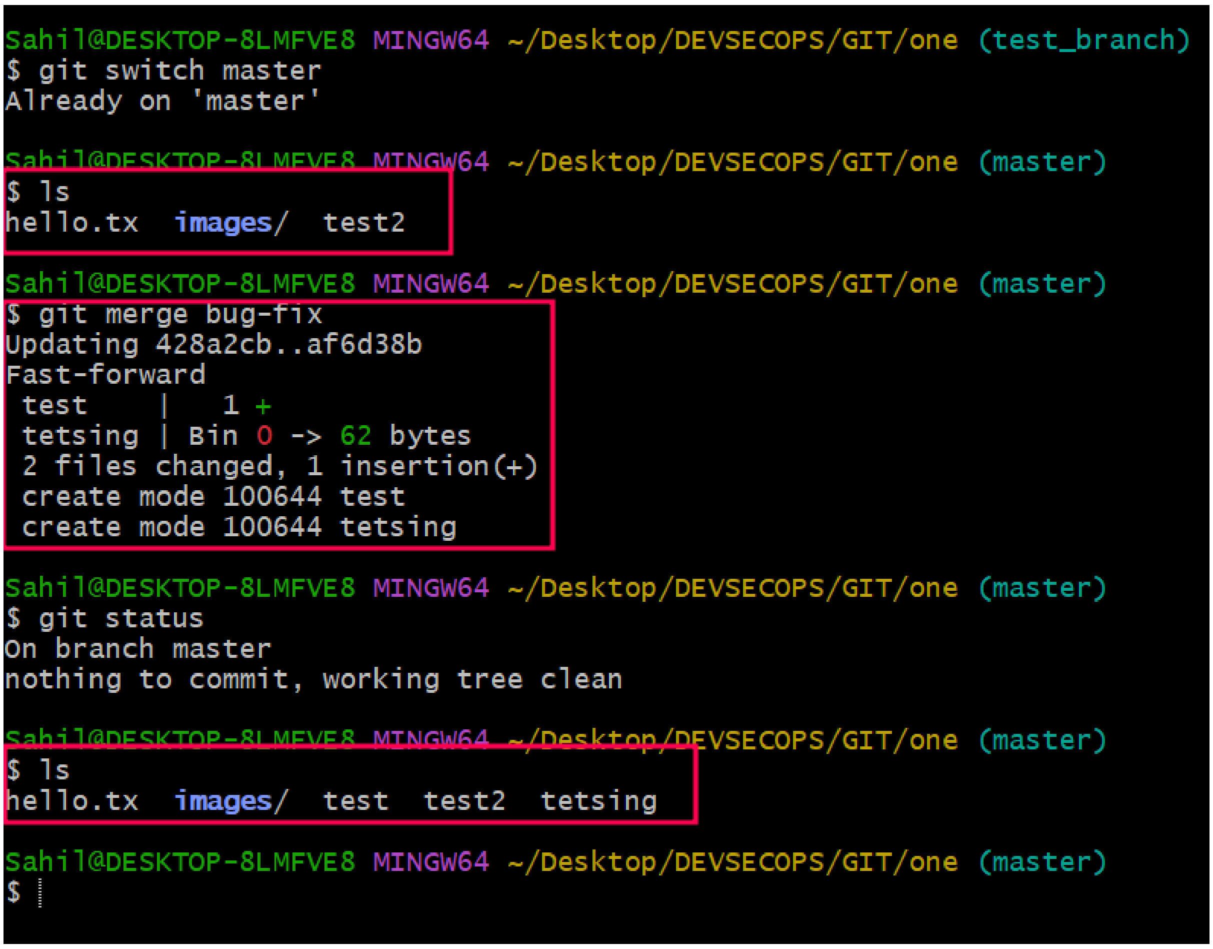
Git merge is a command used to combine the changes from two different branches into one. It helps in integrating work from multiple branches, ensuring that all changes are integrated and up-to-date in the target branch without losing any progress made on either branch.

- Merging is about bringing changes from one branch to another.
- In Git we have two types of merges :
    - Fast-Forward Merges (If branches have not diverged)
    - 3-Way Merges (if branches have diverged)

FAST FORWARD MERGE

`git switch master`

`git merge test_branch`



```
Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (test_branch)
$ git switch master
Already on 'master'

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ ls
hello.tx  images/  test2

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git merge bug-fix
Updating 428a2cb..af6d38b
Fast-forward
 test    |   1 +
 tetsing | Bin 0 -> 62 bytes
 2 files changed, 1 insertion(+)
 create mode 100644 test
 create mode 100644 tetsing

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git status
On branch master
nothing to commit, working tree clean

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ ls
hello.tx  images/  test  test2  tetsing

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$
```

Rename a branch

You can rename a branch using the following command:

`git branch -m <old-branch-name> <new-branch-name>`

Delete a branch

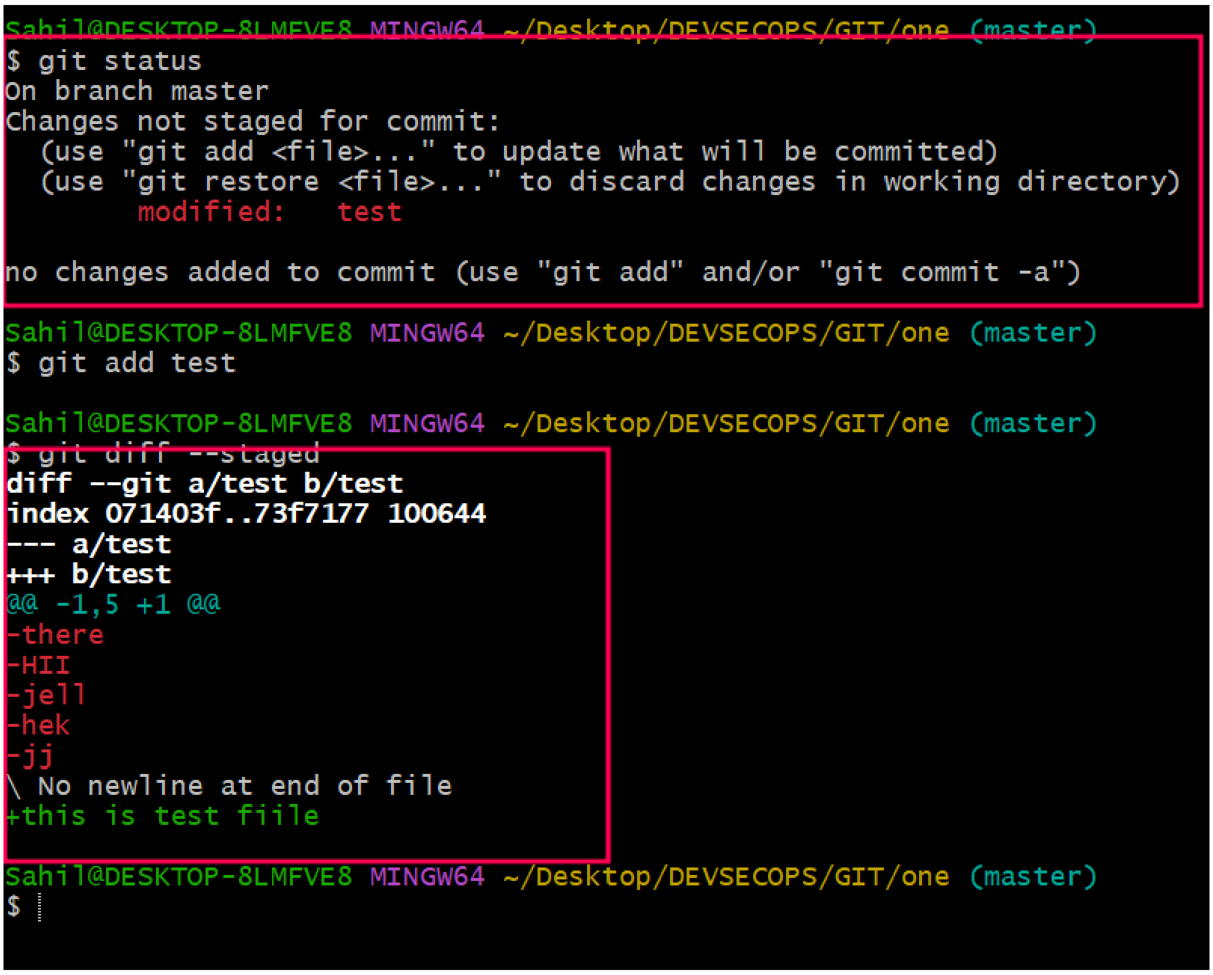
You can delete a branch using the following command:

`git branch -d <branch-name>`

GIT DIFF,STASH AND TAG

The `git diff` is an informative command that shows the differences between two commits. It is used to compare the changes made in one commit with the changes made in another commit. Git consider the changed versions of same file as two different files. Then it gives names to these two files and shows the differences between them.

`git diff:` this is used to see the difference between two commits or branches or stages , at two given time.



```
Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test

no changes added to commit (use "git add" and/or "git commit -a")

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git add test

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$ git diff --staged
diff --git a/test b/test
index 071403f..73f7177 100644
--- a/test
+++ b/test
@@ -1,5 +1 @@
-there
-HII
-jell
-hek
-jj
\ No newline at end of file
+this is test fiile

Sahil@DESKTOP-8LMFVE8 MINGW64 ~/Desktop/DEVSECOPS/GIT/one (master)
$
```

Comparing Staging Area with Repository

`git diff --staged`

This command shows the changes between your last commit and the staging area (i.e., changes that are staged and ready to be committed).

Comparing Two Branche

`git diff <branch-name-one> <branch-name-two>`

This command compares the difference between two branches.

Another way to compare the difference between two branches is to use the following command:

`git diff branch-name-one..branch-name-two`

Comparing Specific Commits:

`git diff <commit-hash-one> <commit-hash-two>`

This command compares the difference between two commits.

Git Stash

Git stash lets you **save your changes temporarily** so you can switch branches or do something else without losing work.

If you have uncommitted changes, Git may not let you switch branches.

To avoid this, use `git stash` to save your work and come back later.

Save your changes:

```
git stash
```

Saves all tracked file changes (like modified files).

Untracked files are not included by default.

Save with a name:

```
git stash push -m "WIP: login feature"
git stash save "test"
```

This helps you remember what the stash is for.

(Note: `git stash save` is older, use `push -m` instead.)

Save untracked files too:

```
git stash -u
```

Adds untracked files (like new files not added to Git yet).

See all stashes:

```
git stash list
```

Shows all your saved stashes.

Apply the latest stash:

```
git stash apply
```

Applies the most recent stash but keeps it in the stash list.

Apply a specific stash:

```
git stash apply stash@{0}
```

Use the list to find the stash number (e.g., `stash@{0}`).

Apply and remove the stash:

```
git stash pop
```

This applies the stash and removes it from the list.

Delete a specific stash:

```
git stash drop stash@{0}
```

Removes one stash without applying it.

Clear all stashes:

```
git stash clear
```

Deletes all saved stashes.

Use stash on a different branch:

1. Switch to the branch:

```
git checkout <branch-name>
```

1. Apply the stash:

```
git stash apply stash@{0}
```

Git Tags

Tags are used to **mark a specific point (commit)** in your Git history.

They are helpful to remember versions like releases (v1.0, v2.1, etc).

Think of them as **labels** you stick on a commit.

Create a simple tag:

```
git tag <tag-name>
```

This tags the **current commit**.

Create an annotated tag (with message):

```
git tag -a <tag-name> -m "Release 1.0"
```

Annotated tags are recommended because they include your name, date, and message.

List all tags:

```
git tag
```

Shows all the tags in your project.

Tag a specific commit:

```
git tag <tag-name> <commit-hash>
```

Use this if you want to tag an older commit (use `git log` to get the hash).

Push a tag to remote:

```
git push origin <tag-name>
```

This sends your tag to GitHub or any remote repo.

Push all tags to remote:

```
git push --tags
```

Useful to send all tags at once.

Delete a local tag:

```
git tag -d <tag-name>
```

Deletes the tag only from your local machine.

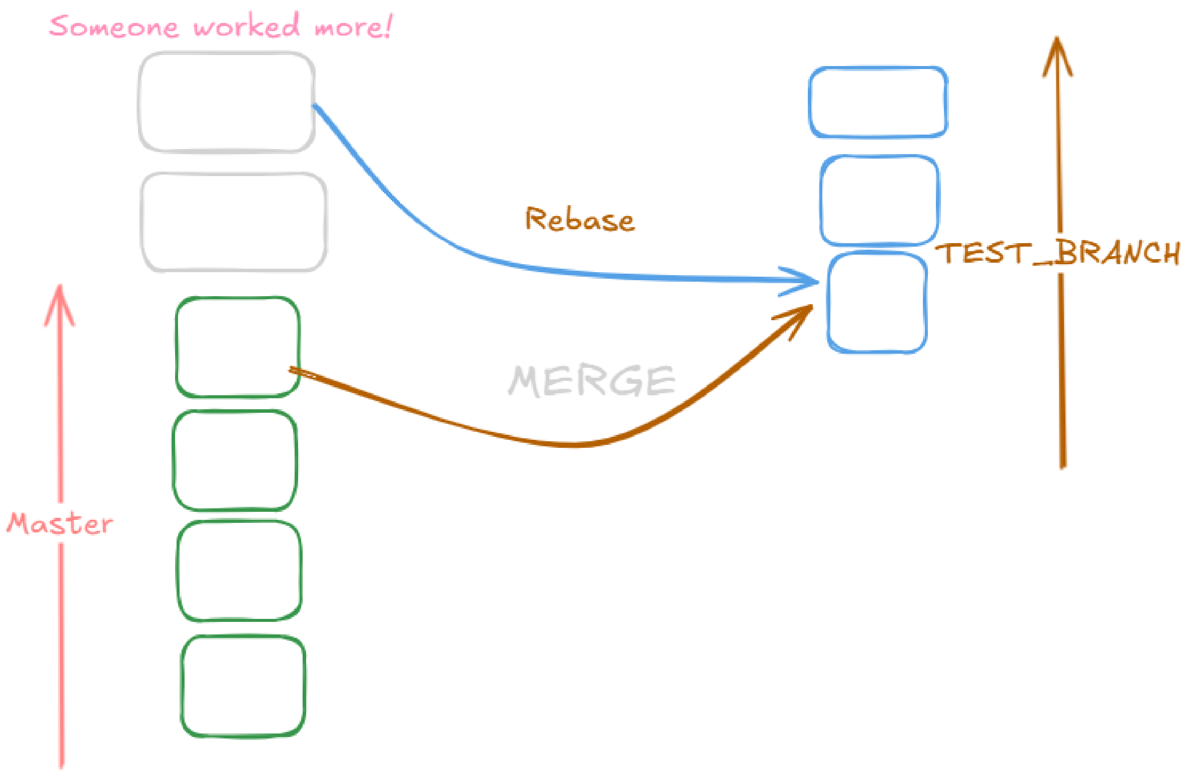
Delete a tag from remote:

```
git push origin :<tag-name>
```

This removes the tag from GitHub or other remote.

GIT REBASE

Rebase rewrites the history!



So in clear words in rebase we just change the base of our last commit in main branch or any another branch to the one in which we want to merge the data. It is helpful in removing unwanted commits .

GIT REFLOG

IT KEEP REFERENCE RECORD FOR VIEWING THE HISTORY OF COMMITS AND THE CHANGES MADE BY US IN THE REPOSITORY!

The reflog is a record that maintains a chronological history of significant changes made to the HEAD, branches, and tags. These changes consist of:

- Commits
- Checkouts
- Merges
- Rebases

Whenever a reference is updated (e.g., a new commit is added to a branch, or you switch to a different branch), Git records this change in the reflog. Each entry in the reflog contains information about the action performed, a timestamp, and the old and new values of the reference.

Recover Lost Commits or Branches with the Reflog

As mentioned previously, one of the most common use cases for the reflog is to recover lost commits or branches. If you accidentally delete a branch or reset it to an unintended commit, you can use the reflog to find the commit SHA-1 hash associated with the branch before the change and then recreate the branch or restore it to the desired state.

As an example, let's go through the entire process to recover a lost branch called "new-feature" that you accidentally deleted or lost track of.

Step 1: Run `git reflog` to view the history of recent reference updates.

Step 2: Go through the reflog entries to find the reference related to the branch you want to recover (in this case, "new-feature"). Each entry should have a description of the action and a commit hash.

The entry related to the branch might look something like this:

```
<commit hash> HEAD@{7}: checkout: moving from main to new-feature
```

Take note of the commit hash associated with the creation of the branch as you will need it for the final step.

Step 3: Now that you have found the commit hash, you have two options:

1. Create a new branch with the same information by typing `git checkout -b new-branch-for-feature <commit hash>` .
2. Restore the existing branch by typing `git branch -f new-feature <commit hash>`. This command will either create a new branch at the specified commit or reset the current branch to that commit.

If you run the `git branch` command, you should see your branch now

The reflog helped you find the commit where the branch was created or last existed, and you used that information to recreate or restore the branch.

Recover lost commits or changes

If you accidentally deleted a branch or made changes that are no longer visible in the commit history, you can often recover them using the reflog. First, find the reference to the commit where the branch or changes existed, and then reset your branch to that reference.

`git reflog <commit-hash>git reset --hard <commit-hash>`

or you can use `HEAD@{n}` to reset to the nth commit before the one you want to reset to.

`git reflog <commit-hash>git reset --hard HEAD@{1}`

GITHUB

What is Github?

Github is a web-based Git repository hosting service. It is a popular platform for developers to collaborate on projects and to share code. Github provides a user-friendly interface for managing and tracking changes to your code, as well as a platform for hosting and sharing your projects with others.

Some other alternative of Github are:

- Gitlab
- Bitbucket
- Azure Repos
- Gitea

configuring GIT:

```
git config --global user.email "your-email@example.com"
git config --global
user.name "Your Name"
```

```
git config --list
```

Setup SSH Key:

follow this instruction to setup ssh keys.

Publish Code to Remote Repository

Now that you have setup your ssh key and added it to your github account, you can start pushing your code to the remote repository.

`git init`

`git add <files>`

`git commit -m "commit message"`

Remote URL Setting

we can check the remote url setting by running the following command:

`git remote -v`

Add Remote Repository

we can add a remote repository by running the following command:

`git remote add origin <remote-url>`

Pushing Code

> `git push remote-name branch-name`

Here `remote-name` is the name of the remote repository that you want to push to and `branch-name` is the name of the branch that you want to push.

`git push origin main`

Setup an upstream remote

Setting up an upstream remote is useful when you want to keep your local repository up to date with the remote repository. It allows you to fetch and merge changes from the remote repository into your local repository. By setting upstream the commits we make in our original offline repo will be directly pushed by using git push . To set up an upstream remote, you can use the following command:

`git remote add upstream <remote-url>`

or `git remote add -u <remote-url>`

or `git remote --set-upstream <remote-url>`

do upstream while pushing our code : git push -u origin main