



GIT & GITHUB NOTES

Ashirbad Swain

GIT

- ✓ Git is system for tracking changes in files and coordinating work in teams at the same workplace.
- ✓ Git is an **open-source distributed revision control** and source code management system.
- ✓ It was initially developed for **Linux Kernel Development** and was created by **Linus Torvalds** in 2005.
- ✓ Basically, Git is a free Software & code Distribution Platform.

Advantages/Features of Git

- ✓ Free and open source
- ✓ Fast and small
- ✓ Implicit backup
- ✓ Security
- ✓ No need of powerful hardware
- ✓ Easier branching
- ✓ Scalable
- ✓ Distributed
- ✓ Speed



Open Source

Git is an open source tool means it free to all of its user.

Scalable

Git is scalable, which means when the number of users increases, the Git can easily handle such situations.

Distributed

Distributed means that instead of switching the project to another machine, we can create a “**clone**” of the entire repository.

Security

Git is secure. It uses **SHA1 (Secure Hash Function)** to name and identify objects within its repository.

Speed

Git is very **fast**, so it can complete all the tasks in a while. Most of the Git operations are done on the local repository, so it provides a **huge speed**.

Supports non-linear development

Git supports **seamless branching and merging**, which helps in visualizing and navigating a non-linear development.

Branching & Merging

Branching & merging are the great features of Git, which makes it different from the other SCM (Source Code Management) tool. Git allows the **creation of multiple branches** without affecting each other. We can perform tasks like **creation**, **deletion** and **merging** on branches and these tasks take a few seconds only.

Data assurance

The git data model ensures the **cryptographic integrity** of every unit of our project. It provides a **unique commit ID** to every commit through a **SHA algorithm**. We can **retrieve** and **update** the commit by commit ID.

Ashirbad Swain

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. Also, the staging area can be considered as a place where Git stores the changes.

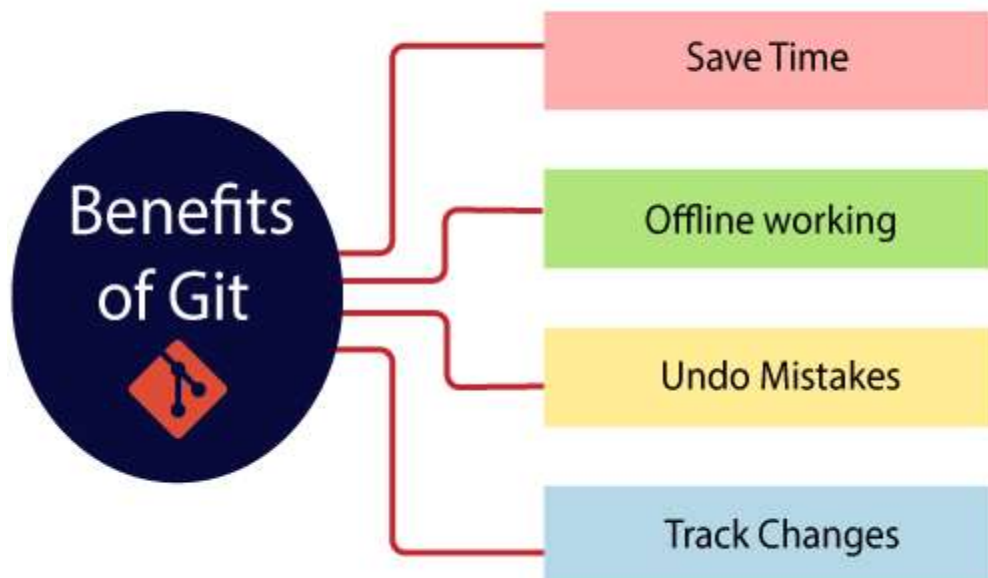
Another feature of Git that makes it apart from other SCM tools is that **it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory**.

Maintain the clean history

Git facilitates with Git rebase; it is one of the most helpful features of Git, it fetches the latest commits form the master branch and put our code on top of that. Thus, it maintains a clean history of the project.

Benefits of Git

Some significant benefits of using git are as follows;



Saves Time

Git is very fast because each command takes only few seconds to execute so we can save a lot of time as compared to login to a GitHub account and find out its features.

Offline Working

Git supports **offline** working. If we are facing internet connectivity issues, it will not affect our work.

Undo Mistakes

One additional benefit of Git is we can **undo** mistakes. Because sometimes the undo can be a savior option for us.

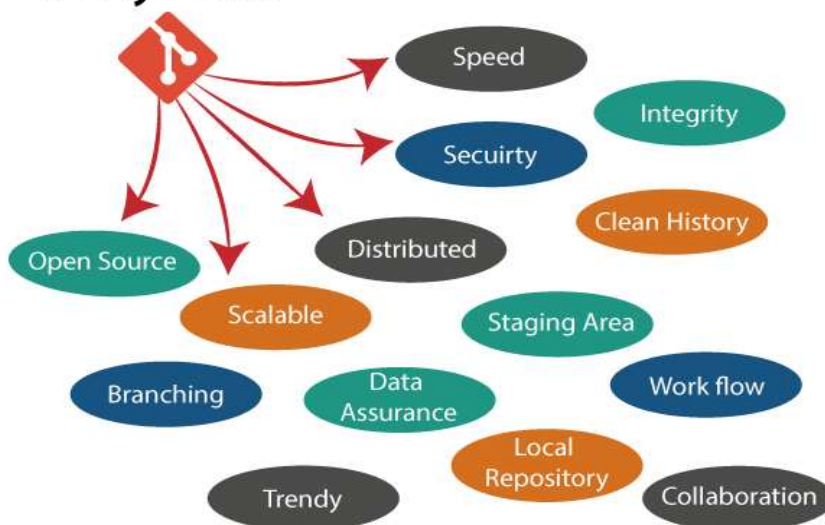
Track the Changes

Git facilitates with some exciting features such as **Diff**, **Log** and **Status** which allows us to track changes so we can **check the status**, **compare** our files or branches.

Why Git?

Because Git has so many **features** and **benefits** that demonstrate undoubtedly Git as the **leading version control system**.

Why Git?



GitHub

GitHub is a **cloud-based hosting service** that lets you manage Git repositories. Means if you have open-source projects that use **Git**, then **GitHub** is designed to help you better **manage** them.

It provides **Web-based Graphical interface**.

It offers both **distributed version control** and **source code management** (SCM) functionality of Git. It also provides some **collaboration features** such as bug tracking, feature requests, task management for every project.



Features of GitHub

GitHub is a place where programmers and designers work together. They collaborate, contribute and fix bugs together.

Some of its features are,

- Collaboration
- Integrated issue and bug tracking
- Graphical representation of branches
- Git repositories hosting
- Project management
- Team management
- Code hosting
- Track and assign tasks
- Conversations

Benefits of GitHub

The key benefits of GitHub are,

- It is easy to contribute to open source projects via GitHub.
- It helps to create an excellent document.
- You can attract recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.
- You can track changes in your code across version.

Difference between Git & GitHub

Git is a version control system that lets you manage and keep track of your source code history. **GitHub** is a cloud-based hosting service that lets you manage **Git** repositories. **Means** if you have open-source projects that use Git, then GitHub is designed to help you better manage them.



Git	GitHub
It is a software	It is a service
It is installed locally on the system	It is hosted on web
It is command line tool	It provides a graphical interface
It is a tool to manage different versions of edits made to files in a git repository	It is a space to upload a copy of the Git repository
It provides functionalities like VCS and SCM	It provides functionalities like VCS, SCM as well as adding few of its own features

Version Control System (VCS)

- ✓ Version Control System is a software that helps software developers to work together and maintain a complete history of their work.
- ✓ **Functions of VCS**
 - Allows developers to work simultaneously
 - Does not allow overwriting each other's changes
 - Maintains a history of every version
- ✓ **Types of VCS**
 - Centralized version control System (CVCS)
 - Distributed/Decentralized version control system (DVCS)
- ✓ Git falls under distributed version control system.

Distributed Version Control System

- ✓ Centralized version control system uses a **central server** to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e. failure of the central server.
- ✓ Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, DVCS comes into picture.
- ✓ **DVCS** clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it.
- ✓ Every check out is a full backup of the repository.
- ✓ Git does not rely on the central server and that is why you can perform many operations when you are offline. You can **commit** changes, **create** branches, **view** logs, and perform other operations when you are offline.
- ✓ You require network connection only to your changes and take the latest changes.

DVCS Terminologies

Local Repository

- ✓ Every VCS tool provides a **private workplace** as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository.
- ✓ Git takes it one step further by providing them a **private** copy of the whole repository. Users can perform many operations with this repository such as **add** file, **remove** file, **rename** file, **move** file, **commit** changes and many more.

Working Directory & Staging Area or Index

- ✓ The working directory is the place where files are **checked out**.
- ✓ In other CVCS, developers generally make modifications and commit their changes directly to the repository.
- ✓ But Git uses a different strategy. Git doesn't track each and every modified file. Whenever you do commit an operation, Git looks for the files present in staging area. Only those files present in **staging area** are considered for commit and not all the modified files.
- ✓ **Workflow of Git**
 - Step 1 - you **modify** a file from the working directory.
 - Step 2 - you **add** these files to the staging area.
 - Step 3 - you perform **commit** operation that moves the files from the staging area. After **push** operation, it stores the changes permanently to the Git repository.

Blobs

- ✓ Blob stands for **Binary Large Object**.
- ✓ Each version of a file is represented by Blob.
- ✓ A blob holds the file data but doesn't contain any metadata about the file.
- ✓ It is binary file, and in Git database, it is named as **SHA1** hash of that file.
- ✓ In Git, files are not addressed by names. Everything is **content-addressed**.

Trees

- ✓ Tree is an object, which represents a directory.
- ✓ It holds blobs as well as other sub-directories.
- ✓ A tree is a binary file that stores references to blobs and trees which are also names as SHA1 hash of the tree object.

Commits

- ✓ Commit holds the current state of the repository.
- ✓ You can consider a commit object as a node of the linked list.
- ✓ A commit is also named by SHA1 hash.
- ✓ Every commit object has a pointer to the parent commit object.

Branches

- ✓ Branches are used to create another line of development.
- ✓ Git has a **master branch**. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch.
- ✓ Every branch is referenced as by **HEAD**, which points to the latest commit in the branch.

Tags

- ✓ Tags assigns a meaningful name with a specific version in the repository.
- ✓ Tags are very similar to branches, but the difference is that tags are **immutable**. It means, tag is a branch, which nobody intends to modify.
- ✓ Once a tag is created for a particular commit, even if you create a new commit, it will not be updated.
- ✓ Usually, developers create tags for **product releases**.

Clone

- ✓ Clone operation creates the instance of the repository.
- ✓ Clone operation not only checks out the working copy, but it also mirrors the complete repository.

Pull

- ✓ Pull operation copies the changes from a remote repository instance to a local one.
- ✓ The pull operation is used for synchronization between two repositories.
- ✓ This is same as **update** operation.

Push

- ✓ Push operation copies changes from a local repository instance to a remote one.
- ✓ This is used to store the changes permanently into the Git repository.
- ✓ This is same as **commit** operation.

HEAD

- ✓ HEAD is a pointer, which always points to latest commit in the branch.
- ✓ Whenever you make a commit, HEAD is updated with the latest commit.
- ✓ The heads of the branches are stored in **.git/refs/heads/directory**.

Revision

- ✓ Revision represents the **version** of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

URL

- ✓ URL represents the **location** of the Git repository.
- ✓ Git URL is stored in **config file**.

Environment Setup

It includes operating system settings, hardware configuration, software configuration, test terminals and other supports to perform the operations. It is an essential aspect of any software.

Git config command

Git supports a command called **git config** that lets you get and set configuration variable that control all facts of how Git looks and operates. It is used to set Git configuration values on a global or local projects level.

Setting **user.name** and **user.email** are the necessary configuration options as your name and email will show up in your commit messages.

Setting username

The user name is used by the Git for each commit.

```
$ git config --global user.name "Ashirbad Swain"
```

Setting email id

The Git uses this email id for each commit.

```
$ git config --global user.email "ashirbad.dkl24@gmail.com"
```

Setting editor

You can set the default text editor when Git needs you to type in a message. If you have not selected any of the editors, Git will use your default system's editor.

To select a different text editor, such as Vim,

```
$ git config --global core.editor Vim
```

Checking Your Settings

You can check your configuration setting; you can use the **git config --list** command to list all the settings that Git can find at that point. This command will list all your settings.

```
$ git config --list
```

Colored output

You can customize your Git output to view a personalized color theme. The **git config** can be used to set these color themes.

```
$ git config --global color.ui true
```

The default value of color.ui is set as auto, which will apply colors to the immediate terminal output stream. You can set the color value as **true**, **false**, **auto** and **always**.

Git configuration levels

The git config command can accept arguments to specify the configuration level.

The following configuration levels are available in the Git config.

1. Local (--local)

It is the default level in Git. Git config will write to a level if no configuration option is given. Local configuration values are stored in **.git/config** directory as a file.

2. Global (--global)

The global level configuration is user-specific configuration. User-specific means, it is applied to an individual operating system user. Global configuration values are stored in a user's home directory.

~/.gitconfig on UNIX systems and C:\user\.gitconfig on windows as a file format.

3. System (--system)

The system-level configuration is applied across an entire system. The entire system means all users on an operating system and all repositories. The system-level configuration file stores in a **gitconfig** file off the system directory.

\$(prefix)/etc/gitconfig on UNIX systems and C:\ProgramData\Git\config on windows.

The **order of priority** of the Git config is local, global and system respectively.

Git Tools

To explore the robust functionality of Git, we need some tools.

Git comes with built-in GUI tools like **git bash**, **git-gui** and **gitk** for committing and browsing. It also supports several third-party tools for users looking for platform-specific experience.

Git Package Tools

GitBash

GitBash is an application for the windows environment. It is used as Git command line for windows. GitBash provides an emulation layer for Git command-line experience.

Bash is an abbreviation of **Bourne Again Shell**.

Bash is a standard default shell on Linux and MacOS. A shell is a terminal application which is used to create an interface with an operating system through commands.

By default, Git windows package contains the GitBash tool. We can access it by right-click on a folder in windows Explorer.

GitBash Commands

GitBash comes with some additional command that are stored in the /user/bin directory of the GitBash emulation and the shell commands like **ssh**, **scp**, **cat**, **find**.

GitBash also includes the full set of Git core commands like **git clone**, **git commit**, **git checkout**, **git push** and many more.

Git GUI

Git GUI is a powerful alternative to GitBash. It offers a graphical version of the Git command line functions, as well as comprehensive visual diff tool. We can access it by simply right click on a folder or location in windows explorer.

Also, we can access it through the command line by typing below command.

```
$ git gui
```

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:

```
$ gitk [git log options]
```

This command invokes the gitk graphical interface and displays the project history.

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.

Some tools are,

1. SourceTree
2. GitHub Desktop
3. Git Extensions
4. GitKraken
5. SmartGit
6. Tower
7. Git Up
8. GitEye
9. Gitg
10. Git2Go
11. GitDrive
12. GitFinder
13. SnailGit
14. Pocket Git
15. Sublime Merge

Git Terminology

Branch: A branch in Git is simply pointer to a commit. The default branch name in Git is **master**.

Checkout: The Git checkout command is used to switch between branches in a repository.

Cherry-Picking: Cherry-picking is meant to apply some commit from one branch into another branch. In case you made a mistake and committed a change into the wrong branch, but do not want to merge the whole branch. You can revert the commit and cherry-pick it on another branch.

Clone: It is used to make a copy of the target repository or clone it.

Fetch: It is used to fetch branches and tags from one or more other repositories, along with the objects necessary to complete their histories.

HEAD: HEAD is the representation of the last commit in the current checkout branch.

Index: The index is a staging area between the working directory and repository means it is used as the index to build up a set of changes.

Master: It's a default branch of Git means "master" is a repository's "default" branch.

Merge: Git merge is used to combine branches.

Origin: It is a reference to the remote repository from a project was initially cloned.

Push Request: The Git push command is used to upload local repository content to a remote repository. This command let others know about changes you've pushed to a branch in a repository.

Rebase: Rebasing is a technique of changing the base of your branch from one commit to another.

Remote Repository: A second copy of a Git repository where you push changes for collaboration or backup.

Repository: Repositories in Git is considered as your project folder. It contains the collection of the files as well as the history of changes made to those files.

Stashing: The Git stash command enables you to switch branch without committing the current branch. Means 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.

Tag: Tags allows you to identify specific release versions of your code. You can think of a tag as a branch that doesn't change. Once it is created, it loses the ability to change the history of commits. There are 2 types of tags are there i.e. annotated and lightweight tag.

Upstream and Downstream: Upstream is where you cloned the repository from (the origin) and Downstream is any project that integrates your work with other works.

Git Revert: The revert command is used to revert some commit. It is an undo type command.

Git Reset: The Git reset command is used to reset the changes. The Git reset command has 3 core forms like, Soft, Mixed and Hard.

Git Ignore: A Git ignore file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected.

Git Diff: Git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data source can be commits, branches, files and more. The git diff command is often used along with git status and git log to analyze the current state of Git repo.

Git cheat sheet: Git cheat sheet that serves as a quick reference for basic Git commands to help you learn Git. Git branches, remote repositories, undoing changes and more.

Git Flow: Git flow is a collection of Git commands. It accomplishes many repository operations with just single commands.

Git Squash: Git squash is an excellent technique to group specific changes before forwarding them to others. You can merge several commits into a single commit.

Git Rm: The term rm stands for remove. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index.

Git Fork: Git fork means you just create a copy of the main repository of a project source code to your own GitHub profile. Then you make your changes and create a pull request to main repository branch, if the main repository owners like your changes they will merge it to the main repository.

Git Command line

Git supports many command-line tools and graphical user interfaces.

Git config command

This command configures the user. This command sets the author name and email address to be used with your commits.

```
$ git config --global user.name "ashu_1998"
$ git config --global user.email ashirbad@gmail.com
```

Git init command

This command is used to create a local repository. This init command will initialize an empty repository.

```
$ git init Demo
```

Git clone command

This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows creating a local copy of that repository on your local directory from the repository URL.

```
$ git clone URL
```

Git add command

This command is used to add one or more files to staging area.

To add one file:

```
$ git add Filename
```

To add more than one file:

```
$ git add*
```

Git commit command

Commit command is used in two scenarios. They are as follows.

Git commit -m: This command changes the head. It records or snapshots the file permanently in the version history with a message.

```
$ git commit -m "Commit Message"
```

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 commit -a
```

Git status command

The status command is used to display the state of the working directory and the staging area.

```
$ git status
```

Git push command

It is used to upload local repository content to a remote repository.

Git push command can be used as follows,

Git push origin master

This command sends the changes made on the master branch, to your remote repository.

```
$ git push [variable name] master
```

```
Ex: $ git push origin master
```

Git push -all

This command pushes all the branches to the server repository.

```
$ git push --all
```

Git pull command

Pull command is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory.

```
$ git pull URL
```

Git branch command

This command lists all the branches available in the repository.

```
$ git branch
```

Git merge command

This command is used to merge the specified branches history into the current branch.

```
$ git merge BranchName
```

```
Ex: $ git merge master
```

Git log command

This command is used to check the commit history.

```
$ git log
```

By default, if no argument passed, Git log shows the most recent commits first. We can limit the number of log entries displayed by passing a number as an option, such as (-3) to show only the last three entries.

```
$ git log -3
```

Git remote command

Git remote command is used to connect your local repository to the remote server. This command allows you to create, view and delete connections to other repositories.

```
$ git remote add origin https://github.com/ashu1998/GitExample1
```

Git flow is the set of guidelines that developers can flow while using Git. These are not rules, it is a standard for ideal project, so that a developer would easily understand the things.

There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches,

- ## Main Branches

1. Master
2. Develop



Master Branch

The master branch is the main branch of the project that contains all the history of final changes. The master branch contains the source code of HEAD that always reflects a final version of the project.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

Develop Branch

It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a “**integration branch**”.

Supportive Branches

The supportive branches are,

1. Feature branches
2. Release branches
3. Hotfix branches

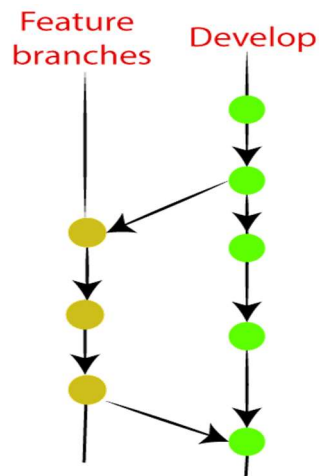
Feature Branch

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.

Release Branch

A release branch is created from develop. The release branch is created for the support of new version release.

Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The released branch should be deployed to a staging server for testing.



Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. Some usual standard of the release branch are as follows,

- Generally, senior developers will create a release branch.
- The release branch will contain the predetermined amount of the feature branch.
- The release branch should be deployed to a staging server for testing.
- Any bugs that need to be improved must be addressed at the release branch.
- The release branch must have to be merged back into developing as well as the master branch.
- After merging, the release branch with the develop branch must be tagged with a version number.

Hotfix Branch

Hotfix branches are similar to release branches; both are created for new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.

Git init

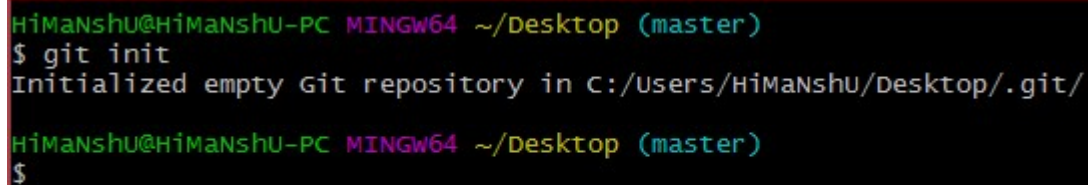
The git init command is used to **create a new blank repository**.

The git init command creates a **.git subdirectory** in the current working directory. This newly created subdirectory contains all of the necessary metadata. These metadata can be categorized into **objects**, **refs** and **temp** files.

It also initializes a HEAD pointer for the master branch of the repository.

Create a Repository for a Blank (New) Project

```
$ git init
```

A terminal window screenshot showing the execution of the 'git init' command. The prompt is 'HiManshu@HiManshu-PC MINGW64 ~/Desktop (master)'. The command '\$ git init' is entered, and the output is 'Initialized empty Git repository in C:/Users/HiManshu/Desktop/.git/'. The prompt then changes to '\$'.

The above command will create an empty **.git** repository.

To **create** a file, run the **cat** or **touch** command as follows,

```
$ touch <file name>
```

To **add** files to the repository, run the git add command,

```
$ git add <file name>
```

We can list all the untracked files by git status command,

```
$ git status
```

Create a Repository and Directory Together

The git init command allows us to create a new blank repository and a directory together.

```
$ git init NewDirectory
```

Git add command

The git add command is used to add files contents to the Index (Staging area).

Git add files

It adds files to the staging area.

```
$ git add <File name>
```

We have created a file for our newly created repository in NewDirectory. To create a file, use the touch command as follows,

```
$ touch newfile.txt
```

And check the status whether it is untracked or not by git status command as follows,

```
$ git status
```

The above command will display the untracked files from the repository. These files can be added to our repository.

So, to add the untracked file, run the below command,

```
$ git add newfile.txt
```

Git add all

We can add more than one files in Git,

```
$ git add -A
```

Or

```
$ git add.
```

The above command will add all the **untracked** files available in the repository.

Removing Files from the staging area

The git add command is also used to remove files from the staging area. If we delete a file from the repository, then it is available to our repository as an untracked file.

Suppose, we have deleted the **newfile3.txt** from the repository. The status of the repository after deleting the file is as follows,

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
on branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   newfile.txt
    new file:   newfile1.txt
    new file:   newfile2.txt
    new file:   newfile3.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
)
    deleted:    newfile3.txt
```

To remove it from the index, run the below command,

```
$ git add newfile3.txt
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add newfile3.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
on branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   newfile.txt
    new file:   newfile1.txt
    new file:   newfile2.txt
```

Add all new and updated files only

Git allows us to stage only updated and newly created files at once. We will use the ignore removal option to do so.

```
$ git add --ignore --removal
```

Add all modified and deleted files

To stage all modified and deleted files only, run the below command,

```
$ git add -u
```

Add files by Wildcard

Git allows us to add all the same pattern files at once. It is another way to add multiple files together.

Suppose I want to add all java files or text files, then we can use pattern .java or .txt.

```
$ git add* .java
```

```
$ git add* .txt
```

Git Undo add

We can undo a git add operation. It can do via git reset command.

```
$ git reset <file name>
```

Git commit

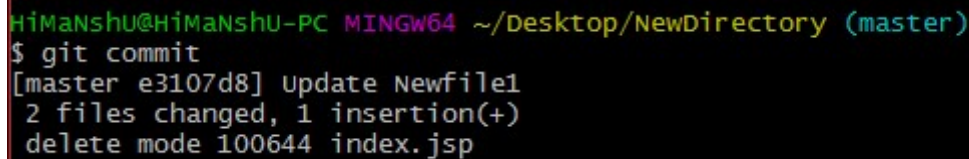
It is used to record the changes in the repository. It is the next command after the git add command.

Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version.

Two different commits will never overwrite because each commit has its own **commit-id**. This commit-id is a cryptographic number created by **SHA (Secure Hash Algorithm)** algorithm.

```
$ git commit
```

The above command will prompt a default editor and ask for a commit message. We have made a change to newfile1.txt and want it to commit it.



```
HIMANSHU@HIMANSHU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git commit
[master e3107d8] Update Newfile1
2 files changed, 1 insertion(+)
delete mode 100644 index.jsp
```

As we run the command, it will prompt a default text editor and ask for a **commit message**.

Then press the **Esc** key and after that **'I'** for insert mode. Type a commit message whatever you want. Press **Esc** after that **':wq'** to save and exit from the editor. Hence, we have successfully made a commit.

Git commit -m: This command changes the head. It records or snapshots the file permanently in the version history with a message.

```
$ git commit -m "Commit Message"
```

The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor.

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.

It will not commit the newly created files.

```
$ git commit -a
```

Git commit amend (change commit message)

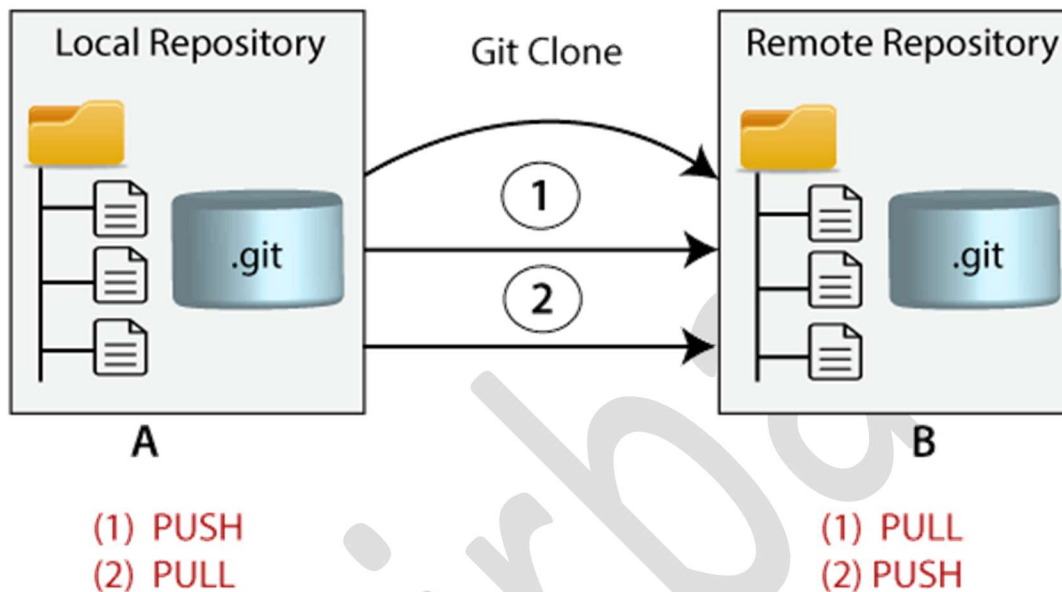
The amend option lets us to edit the last commit. If we accidentally, we have committed a wrong commit message. Then it helps you to correct it.

```
$ git commit -amend
```

```
# Ammend the last commit message.  
# Push the changes to remote by force.  
# USAGE: gamend "Your New Commit Msg"  
function gamend() {  
    git commit --amend -m "$@"  
    git push --force-with-lease  
}
```

Git clone

In Git, cloning is the act of making a copy of any target repository. The target repository can be remote or local. You can clone your repository from the remote repository to create a local copy on your system and also can sync between the two locations.



```
$ git clone <repository URL>
```

Example:

```
$ git clone https://github.com/ashu1998/example.git
```

Cloning a repository into a specific local folder

Git allows cloning the repository into a specific directory without switching to that particular directory.

```
$ git clone https://github.com/ashu1998/example.git "new folder"
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"
Cloning into 'new folder(2)'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

Git has another transfer protocol called **SSH (Secure Shell)** protocol. The above example uses the `git://` protocol, but you can also use `http(s)://` or `user@server:/path.git`, which uses the SSH transfer protocol.

Git Clone Branch

Git allows making a copy of only particular branch from a repository.

To make a clone branch, you need to specify the branch name with `-b` command.

```
$ git clone -b <Branch name> <Repository URL>
```

Example:

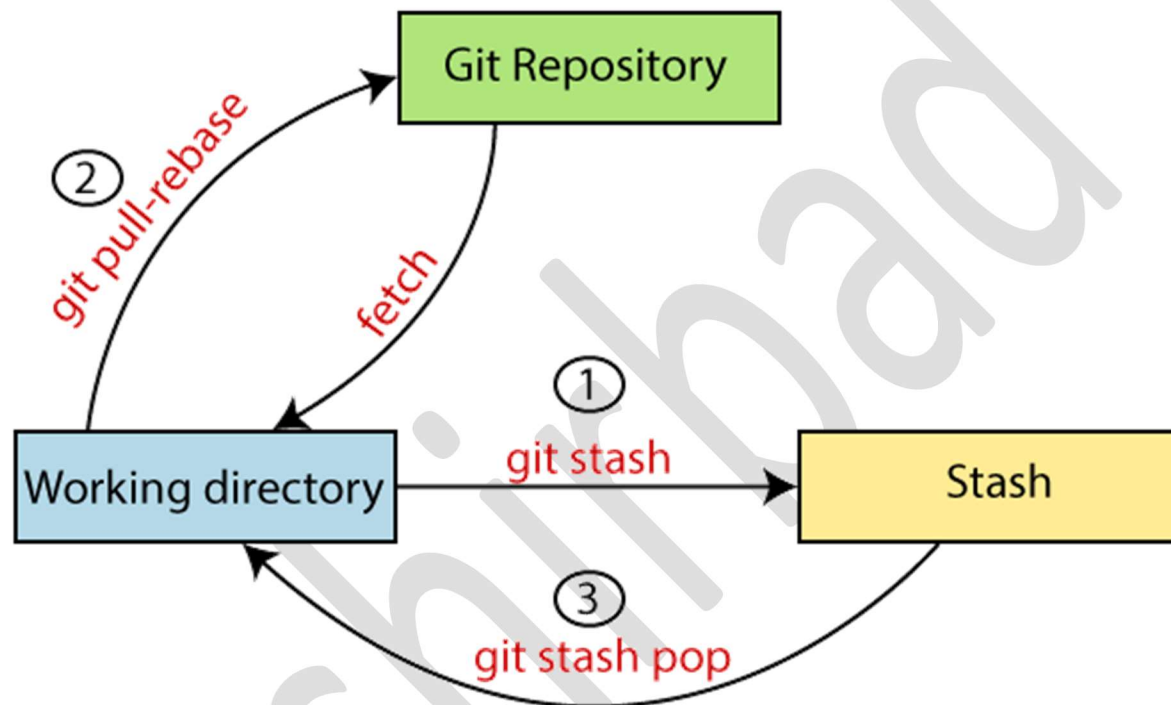
```
$ git clone -b master https://github.com/ashu1998/example.git "new folder"
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ |
```


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.



Generally, the stash's meaning is “store something safely in a hidden place.”

Stashing Work

Suppose 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. Before stash check the status of the repository,

```
$ git status
```

Output:

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git status
On branch test
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:   design.css
    modified:   newfile.txt

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

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

```
$ git stash
```

Output:

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash
Saved working directory and index state WIP on test: 0a1a475 CSS file
```

Now check the status of the current repository. And check the output,

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git status
On branch test
nothing to commit, working tree clean
```

Git stash save (saving stashes with messages)

In Git, the changes can be stashed with a message.

```
$ git stash save "<stashing message>"
```

Output:

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash save "Edited both files"
Saved working directory and index state On test: Edited both files
```

Git stash list (check the stored stashes)

To check the stored stashes, run the below command,

```
$ git stash list
```

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.

```
$ git stash apply
```

Output:

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash apply
on branch test
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:   design.css
    modified:   newfile.txt

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

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.

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git status
on branch test
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:   design.css
    modified:   newfile.txt

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

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**”

In case of more than one stash, you can use “**git stash apply**” command followed by stash index id to apply particular commit.

```
$ git stash apply <stash id>
```

Output:

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash apply stash@{1}
error: Your local changes to the following files would be overwritten by merge:
       design.css
       newfile.txt
Please commit your changes or stash them before you merge.
Aborting
```

If we don't specify a stash, Git takes the most recent stash and tries to apply it.

Git stash changes

We can track the stashes and their changes. To see the changes in the file before stash and after stash operation, run the below command,

```
$ git stash show
```

The above command will show the file that is stashed and changes made on them.

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash show
design.css | 1 +
newfile.txt | 1 +
2 files changed, 2 insertions(+)
```

The above output illustrates that there are two files that are stashed, and two insertions performed on them.

We can exactly track what changes are made on the file. To display the changed content of the file, perform the below command,

```
$ git stash show -p
```

Git Stash pop (reapplying stashed changes)

Git allows the user to re-apply the previous commits by using git stash pop command. The popping option removes the changes from stash and applies them to your working file

```
$ git stash pop
```

Git stash drop (unstash)

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.

```
$ git stash drop
```

We can also delete a particular stash from the queue. To do so we have to pass the stash id in stash drop command.

```
$ git stash drop <stash id>
```

```
Example: $ git stash drop stash@ {1}
```

Git stash clear

The git stash clear command allows deleting all the available stashes at once.

```
$ git stash clear
```

Git stash Branch

If you stashed some work on a particular branch and continue working on that branch. Then, it may create a conflict during merging. So, it is good to stash work on a separate branch. This below command creates a new branch and transfer the stashed work.

```
$ git stash branch <Branch Name>
```

Git Ignore

In Git, ignore used to specify intentionally untracked files that Git should ignore. It doesn't affect the files that already tracked by Git.

The file system in Git is classified into 3 categories,

1. Tracked

Tracked files are such files that are previously staged or committed.

2. Untacked

Untracked files are such files that are not previously staged or committed.

3. Ignored

Ignored files are such files that are explicitly ignored by git.

These files can be derived from your repository source or should otherwise not be committed. Some **commonly ignored files** are,

- Dependency caches
- Compiled code
- Build output directories, like /bin, /out, /target
- Runtime file generated like .log, .lock or .tmp
- Hidden system files like Thumb.db or .DS_Store
- Personal IDE config files, such as .idea/workspace.xml

Git ignore Files

How to ignore files manually

The **.gitignore** file is a file that contains all the formats and files of the ignored file.

Create a file named .gitignore if you do not have it already in your directory. To create a file, use the command touch or cat.

```
$ touch .gitignore
```

Or

```
$ cat .gitignore
```

The above command will create a .gitignore file on your directory. We can track it on the repository.

Name	Date modified	Type	Size
.git	11/5/2019 11:33 AM	File folder	
New folder	11/5/2019 11:31 AM	File folder	
.gitignore	11/5/2019 11:32 AM	Text Document	1 KB
design.css	10/15/2019 2:07 PM	Cascading Style S...	1 KB
index.jsp	9/19/2019 6:10 PM	JSP File	2 KB
master.jsp	9/19/2019 6:10 PM	JSP File	1 KB
merge the branch	9/20/2019 6:05 PM	File	1 KB
newfile.txt	10/15/2019 2:20 PM	Text Document	1 KB
newfile1.txt	10/15/2019 2:27 PM	Text Document	1 KB
newfile2.txt	11/3/2019 5:22 PM	Text Document	1 KB
README.md	9/19/2019 6:10 PM	MD File	1 KB

Now, **add** the files and directories to the .gitignore file that you want to ignore.

Now, to share it on Git, we have to commit it. The .gitignore file is still now in the staging area, we can track it by git status command.

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   newfile2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
```

Now to stage it, we have to commit it.

```
$ git add .gitignore

$ git commit -m "ignored directory created."
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add .gitignore
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m " ignored directory created"
[test2 9d9470e] ignored directory created
2 files changed, 2 insertions(+)
create mode 100644 .gitignore
```

Ashirbad Swain

Global .gitignore file

As we know that we can create multiple .gitignore files for a project. But git also allows us to create a universal .gitignore file that can be used for the whole project. This file is known as a global .gitignore file.

```
$ git config --global core.excludesfile ~/.gitignore_global
```

How to list the ignored files?

In Git, we can list the ignored files.

```
$ git ls-files -I --exclude-standard
```

Or

```
$ git ls-files --ignore -exclude -standard
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git ls-files -i --exclude-standard
newfile.txt
newfile1.txt
newfile2.txt
```

The above command will list all available ignored files from the repository. In the given command, -I option stands for ignore and --exclude-standard is specifying the exclude pattern.

[Git Fork](#)

A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project. One of the excessive uses of forking is to propose changes for bug fixing.

To solve an issue for a bug that you found, you can,

- Fork the repository
- Make the fix
- Forward a pull request to the project owner.

Forking is not a git function; it is a **feature** of Git like GitHub.

Ashirbad Swain

When to use Git Fork

Following are the reasons for forking the repository,

- Propose change to someone else's project.
- Use an existing project as a starting point.

How to fork a repository?

It is a straight forward process, steps for forking the repository are as follows,

- Login to the GitHub account
- Find the GitHub repository which you want to fork
- Click the fork button on the upper right side of the repository's page.

Fork vs. Clone

Both commands are used to create another copy of the repository. But the significant difference is that the fork is used to create a **server-side copy**, and clone is used to create a **local copy** of the repository.

There is no particular command for forking the repository; instead, it is a service provided by third-party Git service like GitHub. Comparatively, git clone is a command-line utility that is used to create a local copy of the project.

Git Repository

In Git, repository is considered as your **project folder**. A repository has all the project-related data and It contains the collection of the files as well as the history of changes made to those files.

Getting a Git Repository

There are 2 ways to obtain a repository, they are as follows,

- Create a local repository and make it as Git repository
- Clone a remote repository (already exists on a server)

Initializing a Repository

To initialize a new repository, run the below command,

```
$ git init
```

We can list all the untracked files by git status command,

```
$ git status
```

Then to add untracked file, we have to run the below command,

```
$ git add <Filename>
```

To commit a file, perform the git commit command as follows,

```
$ git commit -m "Commit message"
```

Cloning an existing repository

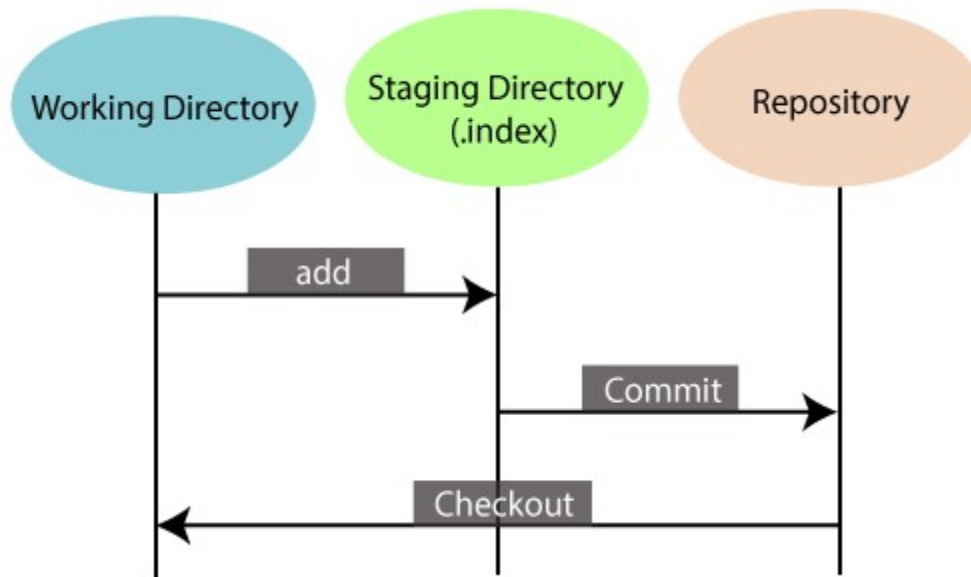
We can clone an existing repository. Also, we can get nearly all data from server with git clone command.

```
$ git clone <Repository URL>
```

```
$ git clone https://github.com/ashu1998/example
```

Git Index

The **Git index** is a staging area between the working directory and the repository. It is used to build up a set of changes that you want to commit together.



There are 3 places in Git where file changes can reside and these are,

- **Working Directory**

When you worked on your project and made some changes, you are dealing with your project's working directory. This project directory is available in your computer's file system. All the changes you make will remain in the working directory until you add them to the staging area.

- **Staging area**

The staging area can be described as a preview of your next commit. The staging area can be considered as a **real area** where git stores the changes.

Staging area means it is currently in the **index**.

- **Repository**

In Git, repository is considered as your **project folder**. A repository has all the project-related data and it contains the collection of the files as well as the history of changes made to those files.

Now, the Git status command allows you to see which files are staged, modified but not yet staged, and completely untracked.

```
$ git status
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   newFile.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

In the given output, the status command shows the index.

As we mentioned earlier index is a file, not a directory, so Git is not storing objects into it. Instead, it stores information about each file in our repository.

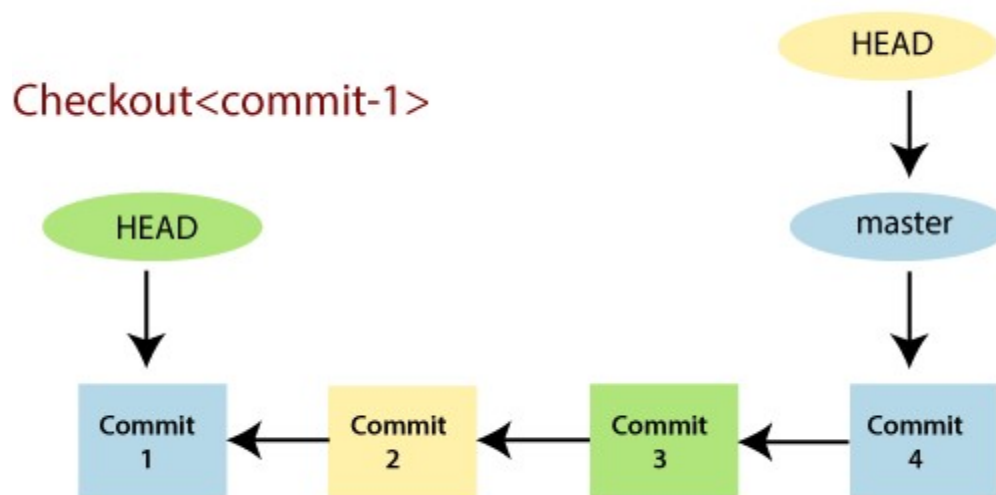
This information could be:

- **mtime**: It is the time of the last update.
- **file**: It is the name of the file.
- **Wdir**: The version of the file in the working directory.
- **Stage**: The version of the file in the index.
- **Repo**: The version of the file in the repository.

And finally, Git creates your working directory to match the content of the commit that HEAD is pointing.

Git HEAD

The HEAD points out the last commit in the current checkout branch. The HEAD can be understood as the “**current branch**”. When you switch branches with ‘checkout’, the HEAD is transferred to the new branch.



Git show HEAD

The Git show HEAD is used to check the status of the HEAD. This command will show the location of the HEAD.

```
$ git show HEAD
```

Git detached HEAD

Detached HEAD mode allows you to discover an older state of a repository. It is a natural state in Git.

The HEAD is capable of referring to a specific revision that is not associated with a branch name. this situation is called a **detached HEAD**.

Or

When HEAD doesn't point to most recent commit, such state is called detached HEAD.

Git origin master

The “**Git origin master**” is used in the context of a remote repository. It is used to deal with the remote repository.

Git master

It's **default** branch of Git. After cloning a project from a remote server, the resulting local repository contains only a single local branch. This branch is called a “**master**” branch.

The master branch is referred to as the main branch. Master branch is considered as the final view of the repo.

Git origin

The **default** remote repository is called origin. Also, origin is referred to the remote repository where you want to publish your comment.

Central Repository



Example:

The URL parameter acts as an origin to the “clone” command for the cloned local repository.

```
$ git clone https://github.com/ashu1998/git-example
```

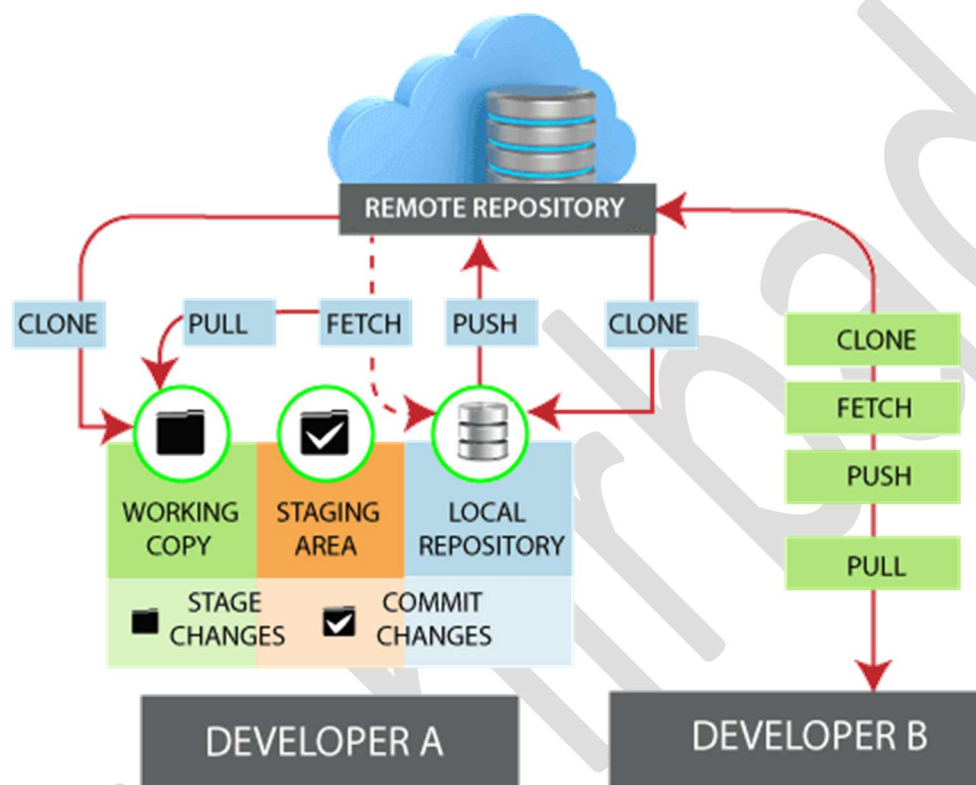
Some commands in which the term **origin** and **master** are widely used are as follows,

- Git push origin master - is used to push the changes to the remote repository.
- Git pull origin master - is used to access the repository from remote to local.

Git remote

The term remote is concerned with the remote repository. It is a shared repository that all team members use to exchange their changes.

The developers can perform many operations with the remote server. These operations can be clone, fetch, push, pull and many more.



Check your remote

To check the **configuration** of the remote server, run the git remote command. It allows accessing the connection between remote and local.

```
$ git remote
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote
origin
```

Git remote -v

Git remote -v is used to show the URLs that git has stored as a short name or shows the available remote connections. These short names are used during the reading and write operation.

Here, -v stands for **verbose**.

```
$ git remote -v
```

Or

```
$ git remote -verbose
```

Git remote add

When we fetch a repository implicitly, git adds a remote for the repository. Also, we can explicitly add a remote for a repository. We can add a remote as a shot nickname or short name. To add remote as a short name, the command will be,

```
$ git remote add <short name><remote URL>
```

A terminal window with a black background and green text. The prompt is 'HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)'. The first command is '\$ git remote add hd https://github.com/ImDwivedi1/hello-world'. The second command is '\$ git remote -v'. The output shows 'hd https://github.com/ImDwivedi1/hello-world (fetch)' and 'hd https://github.com/ImDwivedi1/hello-world (push)'. The 'hd' is highlighted with a blue box. The third command is '\$' and the prompt is 'HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)'.

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$ git remote add hd https://github.com/ImDwivedi1/hello-world

HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$ git remote -v
hd      https://github.com/ImDwivedi1/hello-world (fetch)
hd      https://github.com/ImDwivedi1/hello-world (push)

HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo (master)
$
```

In the above output, I have added a remote repository with an existing repository as a short name “hd”. Now, you can use “hd” on the command line in place of the whole URL.

Fetching and Pulling remote branch

To fetch the data from your remote project, run the below command:

```
$ git fetch <remote>
```

To clone the remote repository from your remote projects, run the below command:

```
$ git clone <remote>
```

To pull the repository, run the below command:

```
$ git pull <remote>
```

Pushing to remote branch

The Git push command is used to share a project or send updates to the remote server. It is used as:

```
$ git push <remote> <branch>
```

To update the main branch of the project, use the below command:

```
$ git push origin master
```

Git Remove remote

To remove a connection, perform the git remote command with remove or rm option. It can be done as:

```
$ git remote rm <destination>
```

Or

```
$ git remote remove <destination>
```

Git remote rename

The command is used to rename the remote server is:

```
$ git remote rename <old name> <new name>
```

Git Show remote

To see additional information about a particular remote, use the git remote command along with show sub-command. It is used as:

```
$ git remote show <remote>
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote show origin
* remote origin
Fetch URL: https://github.com/ImDwivedi1/GitExample2
Push URL: https://github.com/ImDwivedi1/GitExample2
HEAD branch: master
Remote branches:
  BranchCherry      new (next fetch will store in remotes/orig
in)
  PullRequestDemo  new (next fetch will store in remotes/orig
in)
  master            tracked
Local ref configured for 'git push':
  master pushes to master (up to date)
```

Git Change remote (changing a remote's URL)

The **git remote set** command is used to change the URL of the repository

```
$ git remote set-url <remote name><new URL>
```

Example:

```
$ git remote set-url origin https://github.com/URLchanged
```

Git tags

Tags are used to mark a commit stage as relevant. We can tag a commit for a future reference. Primarily, it is used to mark a project's initial point like v1.1.

Tags are much like branches, and they do not change once initiated. We can have any number of tags on a branch or different branches.

There are 2 types tags,

- **Annotated tag**
- **Light-weight tag**

Both of these tags are similar, but they are different in case of the amount **metadata** stores.

When to create a tag

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

Git create tag

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

```
$ git checkout <Branch name>
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (testing)
$ git checkout master
Switched to branch 'master'
```

Now you can tag by using the git tag command. Create a tag with some name say v1.0, v1.1 or any other name.

```
$ git tag <tag name>
```

```
Example: $ git tag projectv1.0
```

Git list tag

We can list the available tags in our repository. There are **three** options that are available to list the tags in the repository.

- Git tag
- Git show
- Git tag -l “. *

Git tag

It is the most generally used option to list all the available tags from the repository. It is used as:

```
$ git tag
```

To show the details of a **particular tag**, the command will be,

```
$ git tag show <tag name>
```

Example: \$ git tag show projectv1.0

Git tag -l “. *

It is also a specific command-line tool. It displays the available tags using wild card pattern. Suppose we have ten tags as v1.0, v1.1, v1.2 up to v1.10. Then we can list all v pattern using tag pattern v.

```
$ git tag -l "<pattern>. *"
```

Example: \$ git tag -l "pro*"

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag -l "pro*"
projectv1.0
projectv1.1
```

Annotated tags

Annotated tags are tags that store extra metadata like developer name, email, date and more. They are stored as a bundle of objects in the Git database.

To create an annotated tag, run the below command:

```
$ git tag <tag name> -m "<tag message>"
```

Example:

```
$ git tag projectv1.1 -m "It's a release point for projectv1.1"
```

Light-Weight tag

Usually, it is a commit stored in a file. It does not store unnecessary information to keep it light-weight. No command-line option such as -a, -s or -m are supplied in light-weighted tag.

```
$ git tag <tag name>
```

Example:

```
$ git tag projectv1.0
```

Git push tag

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

- Git push origin <tag name>
- Git push origin -tags/ Git push --tags

Git push origin

We can push any particular tag by using the git push command. It is used as follows:

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

Example: \$ git push origin projectv1.0

Ashirbad Swain

Git push origin --tag/ Git push --tags

The given command will push all the available tags at once. It is used as follows:

```
$ git push origin --tags
```

```
Or $ git push --tags
```

Git delete tag

Git allows deleting a tag from the repository at any moment. To delete a tag, run the below command:

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

```
Or $ git tag --delete <tag name>
```

```
Example: $ git tag --d projectv1.0
```

Delete a remote tag

We can also delete a tag from the remote server. To delete a tag from the remote server, run the below command:

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

```
Or $ git push origin --delete <tag name>
```

Delete multiple tags

We can delete more than one tag just from a single command. To delete more than one tag simultaneously, run the below command:

```
$ git tag --d <tag1> <tag2>
```

```
Or git push origin --d <tag1><tag2>
```

Git checkout tag

There is no actual concept of check out the tags in git. However, we can do it by creating a new branch from a tag. To check out a tag, run the below command:

```
$ git checkout --b <new branch name> <tag name>
```

The above command will create a new branch with the same state of the repository as it is in the tag.

Create a tag from an older commit

If you want to go back to your history and want create a tag on that point, run the below command:

```
<git tag <tag name> <reference of commit>
```

Suppose I want to create a tag for my older commit, then the process will be as follows:

Check the older commits:

To check the older commit, run the git status command.

```
$ git status
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (new_branchv1.1)
$ git log
commit 56afce0ea387ab840819686ec9682bb07d72add6 (HEAD -> new_branchv1.1, tag: -d, tag: --delete, tag: --d, tag: projectv1.1, origin/master, testing, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css

commit 0a1a475d0b15ecce744567c910eb0d8731ae1af3 (test)
...skipping...
commit 56afce0ea387ab840819686ec9682bb07d72add6 (HEAD -> new_branchv1.1, tag: -d, tag: --delete, tag: --d, tag: projectv1.1, origin/master, testing, master)
```

The above output is showing the older commits. Suppose I want to create a tag for my commit, starting with 828b9628. Copy the particular reference of the commit, and pass it as an argument in the above command.

Like,

```
$ git tag oldversion 828b9628
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (new_branchv1.1)
$ git tag oldversion 828b9628a8

HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (new_branchv1.1)
$ git tag
oldversion
projectv1.1
```

In the above output, an earlier version of the repository is tagged as an oldversion.

Upstream and Downstream

Generally, **upstream** is from where you clone the repository, and **downstream** is any project that integrates your work with other work.

Git set-upstream

The Git set-upstream allows you to set the default remote branch for your current local branch. By default, every pull command sets the master as your default remote branch.

To check the remote server, use the below command:

```
$ git remote -v
```

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git remote -v
origin  https://github.com/ImDwivedil/GitExample2 (fetch)
origin  https://github.com/ImDwivedil/GitExample2 (push)
```

Now check the available branches, run the below command:

```
$ git branch -a
```

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git branch -a
* master
  new_branchv1.1
  test
  test2
  testing
remotes/origin/BranchCherry
remotes/origin/PullRequestDemo
remotes/origin/master
```

To push the changes and set the remote branch as default, run the below command:

```
$ git push --set-upstream origin master
```

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git push --set-upstream origin master
Everything up-to-date
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

We can also set the default remote branch by using the git branch command. To do so, run the below command:

```
$ git branch --set-upstream-to origin master
```

To display default remote branches, run the below command:

```
$ git branch -vv
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git branch -vv
* master          a3a4f45 [origin/master] Removed last lone from the repository
new_branchv1.1 1778984 Revert "new file2 is edited"
test           0a1a475 CSS file
test2          5f8aab1 added Changes
testing        56afce0 Added an empty newfile2
```

The above output is displaying the branches available on the repository. We can see that the default remote branch is specified by highlighted letters.

Undoing changes

Git checkout

The git checkout command is used to **switch** between branches in a repository.

The git checkout command operates upon three different entities which are **files**, **commits** and **branches**.

Operations on Git checkout

We can perform many operations by git checkout command like,

- Switch to a specific branch
- Create a new branch
- Checkout a remote branch etc.

Checkout branch

To demonstrate available branches in repository, use the below command:

```
$ git branch
```

Now, you have the list of branches. To switch between branches, use the below command:

```
$ git checkout <branch name>
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
  TestBranch
* master

HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout testbranch
Switched to branch 'testbranch'
```

Create and switch branch

The git checkout command let you create and switch to a new branch.

The git checkout -b option is a convenience flag that performs run git branch <new-branch> operation before running git checkout <new-branch>.

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

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout -b branch3
Switched to a new branch 'branch3'
A       design.css
```

Checkout remote branch

To checkout a remote branch, you have first to fetch the contents of the branch.

```
$ git fetch --all
```

In the latest versions of Git, you can check out the remote branch like a local branch.

```
$ git checkout <remote branch>
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git fetch --all
Fetching origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/GitExample2
* [new branch]      edited    -> origin/edited

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout edited
Switched to a new branch 'edited'
Branch 'edited' set up to track remote branch 'edited' from 'origin'.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (edited)
$ |
```

In the earlier versions, below command is used to check out the remote branch.

```
$ git checkout <remote branch> origin/ <remote branch>
```

Git Revert

The git revert command is used to apply revert operation or revert some changes. It is an undo type command.

It can be useful for tracking bugs in the project.

```
$ git revert
```

Git revert options

Git revert allows some additional operations like editing, no editing, cleanup, and more. Let's understand these options briefly:

< commit>: The commit option is used to revert a commit. To revert a commit, we need the commit reference id. The git log command can access it.

```
$ git revert <commit-ish>
```

<--edit>: It is used to edit the commit message before reverting the commit. It is a default option in git revert command.

```
$ git revert -e <commit-ish>
```

-m parent-number /--mainline parent-number: it is used to revert the merging. Generally, we cannot revert a merge because we do not know which side of the merge should be considered as the mainline. We can specify the parent number and allows revert to reverse the change relative to the specified parent.

-n/--no edit: This option will not open a text editor. It will directly revert the last commit.

```
$ git revert -n <commit-ish>
```

--cleanup=<mode>: The cleanup option determines how to strip spaces and comments from the message.

-n/--no-commit: Generally, the revert command commits by default. The no-commit option will not automatically commit. In addition, if this option is used, your index does not have to match the HEAD commit.

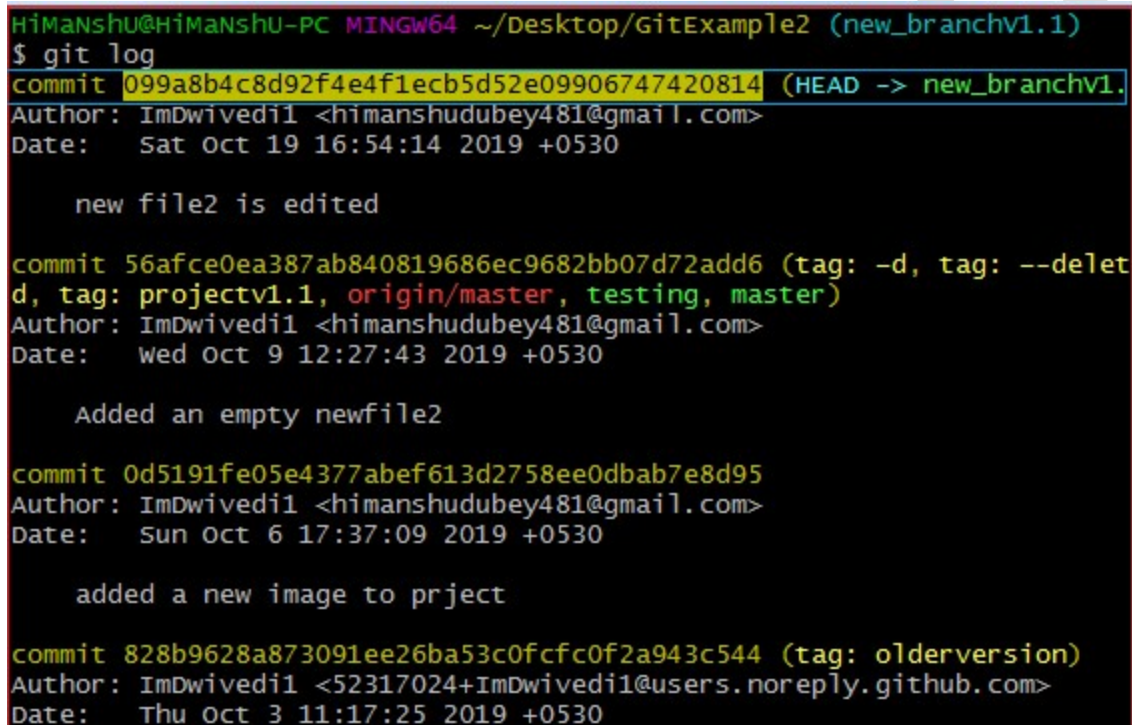
ASIIIBUU SWUUII

Git Revert to previous commit

Suppose you have made a change to a file say newfile2.txt of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now, you want to undo the changes you can do so. Git allows you to correct your mistakes.

To undo changes, but we will need the **commit-ish**. To check the commit-ish, run the below command:

```
$ git log
```



```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (new_branchv1.1)
$ git log
commit 099a8b4c8d92f4e4f1ecb5d52e09906747420814 (HEAD -> new_branchv1.1)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Sat Oct 19 16:54:14 2019 +0530

    new file2 is edited

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: projectv1.1, origin/master, testing, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Sun Oct 6 17:37:09 2019 +0530

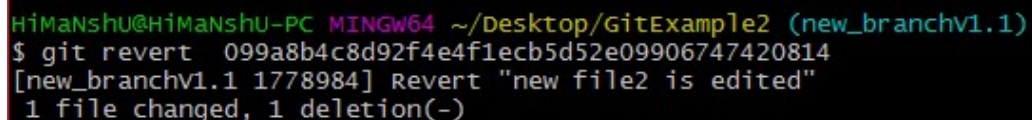
    added a new image to project

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date: Thu Oct 3 11:17:25 2019 +0530
```

Now copy the most recent commit-ish to revert.

```
$ git revert 099a8b4c8d92f4e4f1ecb5d52e0990886896
```

The above command will revert my last commit.



```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/GitExample2 (new_branchv1.1)
$ git revert 099a8b4c8d92f4e4f1ecb5d52e09906747420814
[new_branchv1.1 1778984] Revert "new file2 is edited"
1 file changed, 1 deletion(-)
```

ASHIM BUA SWAMI

Git Revert merge

Suppose I have made some changes to my file **design2.css** on the test and merge it with **test2**.

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge test
Updating f1ddc7c..0a1a475
Fast-forward
 design2.css | 6 +++++
 1 file changed, 6 insertions(+)
 create mode 100644 design2.css
```

To revert a merge, we have to get its reference number. To check commit history, run the below command:

```
$ git log
```

```
HiManshU@HiManshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git log
commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (HEAD -> test2, test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date: Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Sat Sep 28 12:31:30 2019 +0530

    new comit on test2 branch

commit 7fe5e7a8a733e55596db3f49d85e66245f88ddc0
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Sat Sep 28 12:33:18 2019 +0530

    new commit in master branch

commit dfb53648625baf81ab66f70944b2dc8c0fc9ef64
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Fri Sep 27 18:18:53 2019 +0530

    commit2

commit 4fddabb36bf1690bec8a3b6605573ca9cffe00f4
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Fri Sep 27 18:18:14 2019 +0530

    commit1

commit a3644e15993d30c7834b700077d4503c9afb6abd
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Wed Sep 25 14:40:35 2019 +0530
```

From the above output, copy your merging commit that you want to revert and run the below command:

```
$ git revert <commit reference> -m 1
```


Git reset

The git reset command is used to reset the changes. The git reset command has 3 core forms if invocation.

- Soft
- Mixed
- Hard

Git reset hard

```
$ git reset --hard
```

Generally, the reset hard mode performs the below operations:

- It will move the HEAD pointer
- It will update the staging area with the content that the HEAD is pointing.
- It will update the working directory to match the staging area.

Git reset mixed

Generally, the reset mixed mode performs the below operations:

- It will move the HEAD pointer
- It will update the staging area with the content that the HEAD is pointing to.

But it will not update the working directory as git hard mode does.

```
$ git reset --mixed
```

Git Reset head (git reset soft)

It is used to change the position of the head and will move the head to the particular commit.

```
$ git reset --soft <commit-sha>
```

Git reset to commit

```
$ git reset <option> <commit-sha>
```

Ashirbad Swain

Git rm

It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the git index.

Limits of Git rm command:

The git rm is operated only on the current branch. The removing process is only applied to the working directory and staging index trees.

```
$ git rm <file name>
```

Git rm cached

Sometimes you want to remove the files from the Git but keep the files in the local repository. Or you don't want or share file on Git. The cached option is used in this case.

```
$ git rm --cached <file name>
```

Undo the Git rm command

We can undo a git remove command by many ways as follows:

```
$ git reset HEAD
```

Or

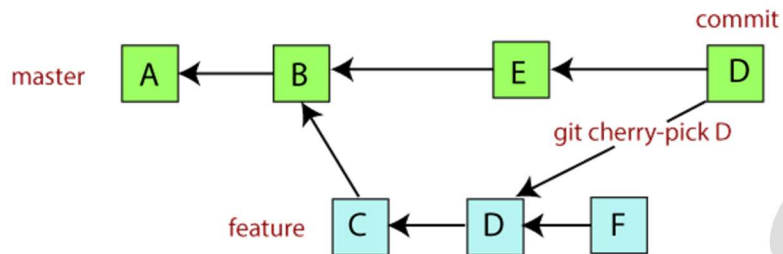
```
$ git reset --hard
```

Also, we can do it by git checkout command,

```
$ git checkout
```

Git Cherry-pick

Cherry-picking in Git stands for applying some commit from one branch into another branch. In case you made a mistake and committed a change into the wrong branch, but do not want to merge the whole. You can revert the commit and apply it on another branch.



Some scenarios in which you can cherry-pick:

- Accidentally make a commit in a wrong branch
- Made the changes proposes by another team member

Syntax:

```
$ git cheery-pick <commit id>
```

Usage of cheery-pick:

- ✓ It is a handy tool for team collaboration.
- ✓ It is necessary in case of bug fixing because bugs are fixed and tested in the development branch with their commits.
- ✓ It is mostly used in undoing changes and restoring lost commits.
- ✓ You can avoid useless conflicts by using git cheery-pick instead of other options.
- ✓ It is useful tool when a full branch merge is not possible due to incompatible versions in the various branches.
- ✓ The git cheery-pick is used to access the changes introduced to a sub-branch, without changing the branch.

Inspecting changes

Git log

Git log is a utility tool to review and read a history of everything that happens to a repository.

Generally, the Git log is a record of commits. A Git log contains the following data:

- ✓ **Commit hash:** which is a 40-character checksum data generated by SHA (Secure Hash Algorithm). It is a unique number.
- ✓ **Commit Author metadata:** the information of authors such as author name and email.
- ✓ **Commit Date metadata:** It's a date timestamp for the time of commit.
- ✓ **Commit title/message:** it is the overview of the commit given in the commit message.

Syntax:

```
$ git log
```

To **exit** the git log is to press the **q** (**Q** for quit).

Git log online

The online option is used to display the output as one commit per line. It also shows the output in brief like the first seven characters of the commit SHA and commit message.

Syntax:

```
$ git log --online
```

Usually we can say that the `--online` flag causes git log to display:

- ✓ One commit per line
- ✓ The first seven characters of the SHA
- ✓ The commit messages

Git log stat

Generally, we can say that the stat option is used to display

- ✓ The modified files
- ✓ The number of lines that have been added or removed
- ✓ A summary line of the total number of records changed
- ✓ The line that have been added or removed

```
$ git log --stat
```

Git log P or Patch

Generally, we can say that the --patch flag is used to display:

- ✓ Modified files
- ✓ The location of the lines that you added or removed
- ✓ Specific changes that have been made

```
$ git log --patch
```

Or

```
$ git log -p
```

Git log graph

Git log command allows viewing your git log as a graph. To list the commits in the form of graph, run the git log command with **--graph** option.

```
$ git log --graph
```

To make the output more specific, you can combine this command with --online option.

```
$ git log --graph --online
```

Filtering in commit history

We can filter the output according to our needs. We can apply many filters like amount, date, author and more on output.

By Amount:

```
$ git log -<n>
```

Example: `$ git log -3`

By Date and Time:

```
$ git log -after = "yy-mm-dd"
```

Example: `$ git log -after = "2019-11-01"`

We can also pass the applicable reference statement like **"yesterday"**, **"1 week ago"**, **"21 days ago"**, and more.

```
$ git log --after = "21 days ago"
```

We can also track the commits between two dates. To track the commits that were created between two dates, pass a statement reference **--before** and **--after** the date.

We want to track the commits between "2019-11-01" and "2019-11-08".

```
$ git log --after = "2019-11-01" --before = "2019-11-08"
```

By Author:

```
$ git log --author = "Author name"
```

Example: `$ git log --author = "Stephen"`

Or `$ git log --author = "@gmail.com"`

By commit message:

```
$ git log --grep = "commit message"
```

Git Diff

It compares the different versions of data structures. It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits etc.

Some scenarios where Git diff is used:

1. Track the changes that have not been staged.
2. Track the changes that have staged but not committed.

To check the staged changes, run the git diff command along with **--staged** option. This will display the changes of already staged files.

```
$ git diff --staged
```

3. Track the changes after committing a file.

To track the changes after committing a file, run the git diff command with HEAD argument. It will run as follows:

```
$ git diff HEAD
```

4. Track the changes between two commits:

```
$ git diff <commit-sha> <commit2-sha>
```

Git Diff Branches

The git diff command allows us to compare different versions of branches and repository. To get the difference between branches, run the git diff command as follows:

```
$ git diff <branch 1> <branch 2>
```

Example: \$ git diff test test2

Git Status

The git status command is used to display the state of the repository and staging area. It allows us to see the tracked, untracked files and changes. This command will not show any commit records or information.

Mostly, it is used to display the state between **Git add** and **Git commit**.

```
$ git status
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Status when an existing file is modified

Check the status when an existing file is modified. To modify file, run the **echo** command as follows,

```
$ echo < "Text"> Filename
```

```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/NewDirectory (master)
$ echo "Add some text "> newfile3.txt

HiManshu@HiManshu-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newfile3.txt

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

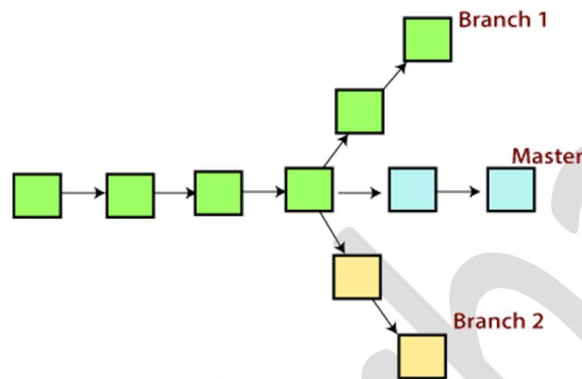
```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   newfile3.txt
```

Branching & Merging

Git Branch

A branch in Git is simply a light-weight movable pointer to one of these commits. A git project can have more than one branch.

When you want to add a new feature or fix a bug, you create a new branch to summarize your changes.



Git master branch

It is the default branch in a git repository. It is initiated when first commit made on the project.

A repository can have only **one master branch**.

Create Branch

```
$ git branch <Branch name>
```

Example: \$ git branch B1

List Branch

```
$ git branch -list OR $ git branch
```


Delete Branch

```
$ git branch -d <Branch name> OR $ git branch -D <Branch name>
```

Example: \$ git branch -d B1

Delete a Remote Branch

```
$ git push origin --delete <Branch name>
```

Example: \$ git push origin --delete Branch2

Switch Branch

```
$ git checkout <Branch name>
```

Example: \$ git checkout branch4

Rename Branch

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

Example: \$ git branch -m branch4 renamedB1

Merge Branch

```
$ git merge <Branch name>
```

Example: \$ git merge renameB1

Switch from master branch

```
$ git checkout <Branch name>
```

Example: \$ git checkout master

Switch to master branch

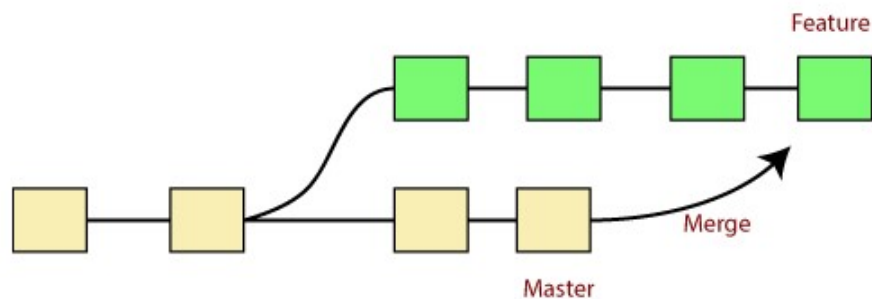
```
$ git checkout <Branch name>
```

Example: \$ checkout master

Git Merge & Merge Conflicts

The **merging** is a procedure to join two or more development history or branch together.

The git merge command facilitates you to take the data created by git branch and integrate them into a single branch.



The git merge command can be used in different scenarios:

1. To merge the specified commit to currently active branch:

```
$ git merge <commit>
```

Set the **commit id** in the place of commit.

2. To merge commits into the master branch:

To merge commits into the master branch, switch over the master branch.

```
$ git checkout master
```

Now, use the git merge command along with master branch name.

```
$ git merge <commit>
```

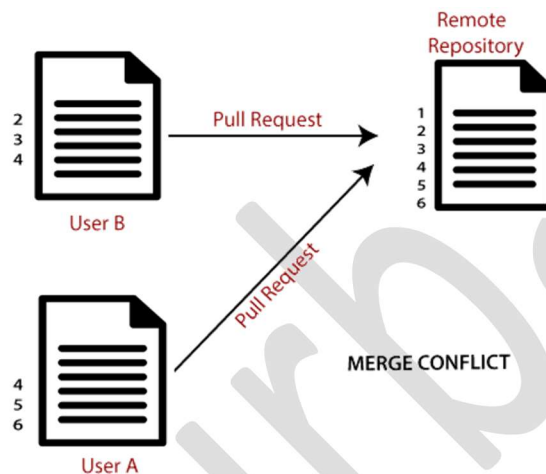
3. Git merge branch:

```
$ git merge <Branch name>
```

Git merge conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called **merge conflict**.

If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**.



Resolve conflict

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict.

```
$ git mergetool
```

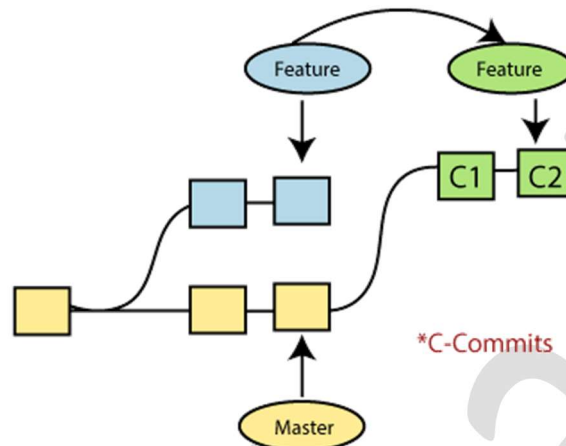
The above command shows the status of the conflict file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the **rebase** command.

It will be used as follows:

```
$ git rebase --continue
```

Git Rebase

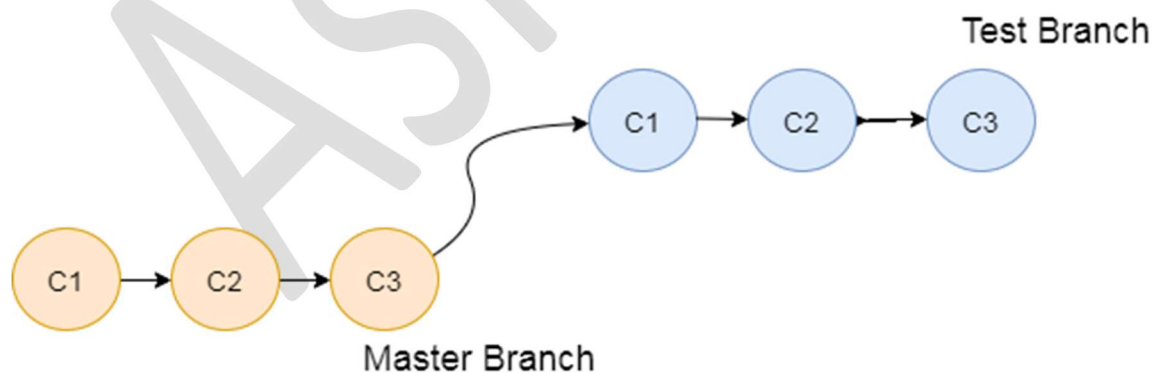
In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit.



Generally, it is an alternative of git merge command. Merge is always a forward changing record. Comparatively, rebase is a compelling history rewriting tool in git. It merges the different commits **one by one**.

Example:

Suppose you have made three commits in your master branch and three in your other branch named test. If you merge this, then it will merge all commits in a time. But if you rebase it, then it will be merged in a linear manner.



How to rebase

```
$ git rebase <branch name>
```

If there are some conflicts in the branch, resolve them and perform below commands to continue changes:

```
$ git status
```

It is used to check the status.

```
$ git rebase --continue
```

The above command is used to continue with the changes you made. If you want to skip the changes, you can skip as follows:

```
$ git rebase --skip
```

When the rebasing is completed. Push the repository to the origin.

Rebase Branch

If we have many commits from distinct branches and want to merge it in one. To do so, we have two choices either we can merge it or rebase it. It is good to rebase your branch.

Syntax:

```
$ git rebase master
```

This command will rebase the mentioned branch and will show as Applying: new commit on test2 branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded test to master.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$
```

Git Interactive Rebase

It is a potent tool that allows various operations like edit, rewrite, reorder and more on existing commits.

Interactive rebase can only be operated on the currently checked out branch. Therefore, set your local HEAD branch at the sidebar.

```
$ git rebase -i
```

When we perform the git interactive command, it will open your default text editor with the above output.

The option it contains are listed below:

Pick (-p):

Pick stands here that the commit is included. Order of the commits depends upon the order of the pick commands during rebase. If you do not want to add a commit, you have to delete the entire line.

Reword (-r):

The reword is quite similar to pick command. The reword option paused the rebase process and provides a chance to alter the commit message. It does not affect any changes made by the commit.

Edit (-e):

The edit option allows for amending the commit. The amending means, commits can be added or changed entirely. We can also make additional commits before rebase continue command. It allows us to split a large commit into the smaller commit; moreover, we can remove erroneous changes made in a commit.

Squash (-s):

The squash option allows you to combine two or more commits into a single commit. It also allows us to write a new commit message for describing the changes.

Ashirbad Swain

Fixup (-f):

It is quite similar to the squash command. It discarded the message of the commit to be merged. The older commit message is used to describe both changes.

Exec (-x):

The exec option allows you to run arbitrary shell commands against a commit.

Break (-b):

The break option stops the rebasing at just position. It will continue rebasing later with '**git rebase --continue**' command.

Drop (-d):

The drop option is used to remove the commit.

Label (-l):

The label option is used to mark the current head position with a name.

Reset (-t):

The reset option is used to reset head to a label.

Git Squash

In Git, the term **squash** is used to squash the previous commits into one. It is not a command; instead, it is a **keyword**.

The **squash** is an excellent technique for group-specific changes before forwarding them to others.

Git Squash commits

Git squash is one of the powerful tools that facilitate efficient and less painful collaboration.

Suppose we have made many commits during the project work, squashing all the commits into a large commit is the right choice than pushing.

Step 1: check the commit history

```
$ git log --online
```

Step 2: choose the commits to squash

```
$ git rebase -i HEAD ~3
```

The above command will open your default text editor and will squash the last three commits.

If we want to merge them into a single commit, then we have to **replace** the word **pick** with the **squash** on the top of the editor. To write on the editor, press 'i' button to enter in **insert mode**. After editing the document, press the **:wq** to save and exit from the editor.

Step 3: update the commits

On pressing **enter** key, a new window of the text editor will be opened to confirm the commit. We can edit the commit message on this screen.

To edit on this editor, press the 'i' button for insert mode and edit the desired text. Press the **:wq** key to save and exit from the editor.

Step 4: push the squash commit

Now, we can push this squashed commit on the remote server. To push this squash commit, run the below command:

```
$ git push origin master
```

Or

```
$ git push -f origin master
```

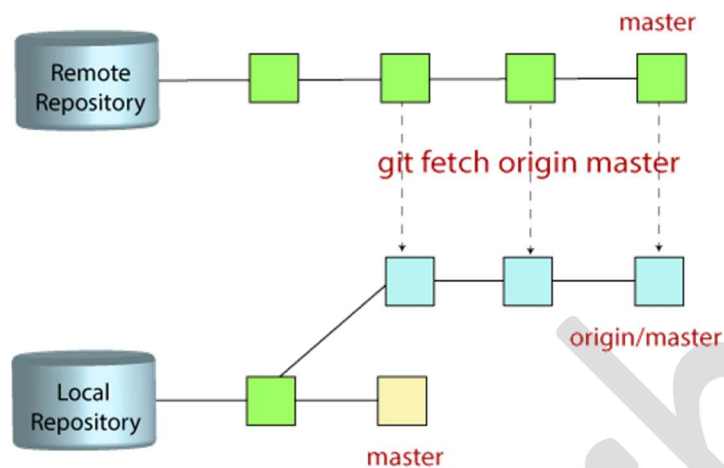
Drawbacks of Squashing

- The squashing commits, and rebasing changes the history of the repository. If any contributor does not pay attention to the updated history, then it may create conflict.
- We may lose granularity because of squashing. Try to make minimum squashes while working with Git.

Collaborating

Git Fetch

Git fetch downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories



Git fetch command

The git fetch command is used to pull the updates from remote-tracking branches.

We can use fetch command with many arguments for a particular data fetch.

1. To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does.

```
$ git fetch <repository URL>
```

2. To fetch a specific branch:

We can fetch a specific branch from a repository. It will only access the element from a specific branch.

```
$ git fetch <branch URL> <branch name>
```

3. To fetch all the branches simultaneously:

The git fetch command allows to fetch all branches simultaneously from a remote repository.

```
$ git fetch --all
```

4. To synchronize the local repository:

Suppose, your team member has added some features to your remote repository. So, to add these updates to your local repository, use the git fetch command.

```
$ git fetch origin
```

Difference between Git fetch and Git pull

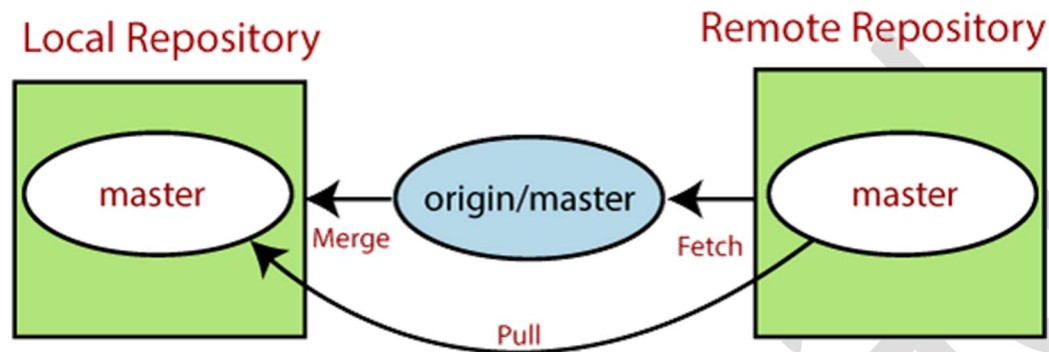
Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository.

So, basically

Git pull = git fetch + git merge

Git Pull / Pull Request

The term is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The Git pull command is used to pull a repository.



Pull request is a process for developers to notify team members that they have completed a feature. Pull request announces all the team members that they need to view the code and merge it into the master branch.

Syntax:

```
$ git pull <option> [<repository URL><refspec>.....]
```

<option>: options are the commands; these commands are used as an additional option in a particular command. Options can be **-q (quiet)**, **-v (verbose)**, **-e (edit)** and more.

<repository URL>: repository URL is your remote repository's URL where you have stored original repositories like GitHub.

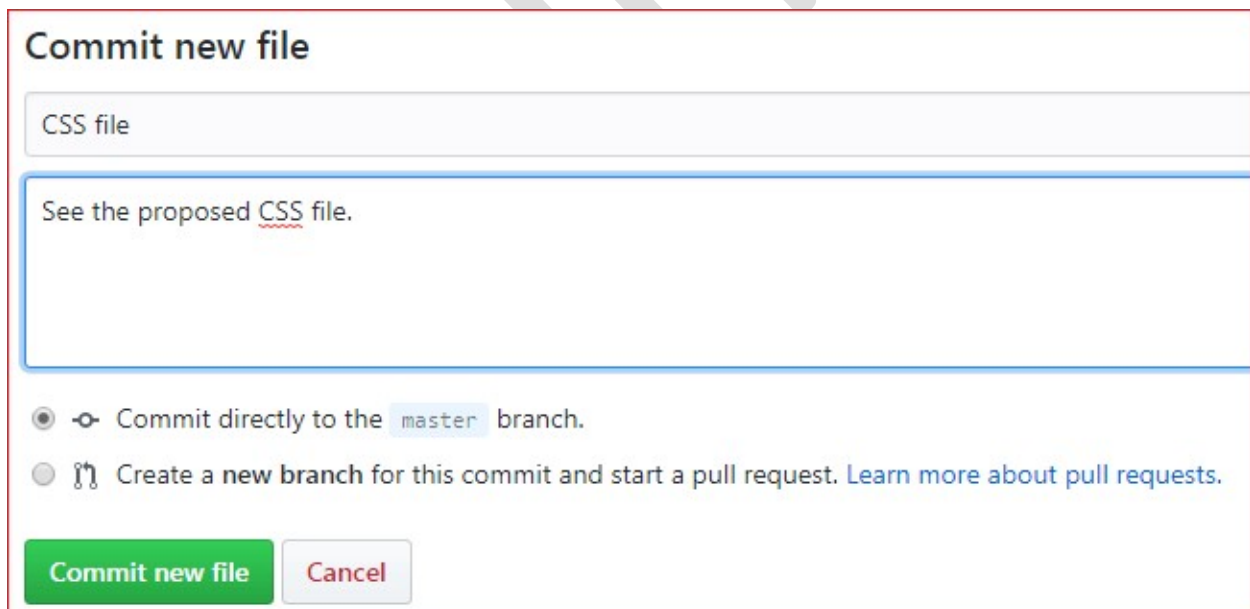
<refspec>: a ref is referred to commit, for example, head (branches), tags, and remote branches. You can check head, tags and remote repository in **.git/ref** directory on your local repository on your local repository. **Refspec** specifies and updates the refs.

How to use pull

To create the file first, go to create a file option given on repository sub-functions. After that, select the file name and edit the file as you want.



Go to the bottom of the page, select a commit message and description of the file. Select whether you want to create a new branch or commit it directly in the master branch.



Now, we have successfully committed the changes.

To pull these changes in your local repository, perform the git pull operation on your cloned repository.

Ashirbad Swain

Default Git Pull

We can pull a remote repository by just using the git pull command. It's default option

```
$ git pull
```

Git Pull Remote Branch

```
$ git pull <remote branch URL>
```

Git Force Pull

The force pull is used for overwriting files. If we want to discard all the changes in the local repository, then we can overwrite it by forcefully pulling it.

Step 1: Use the git fetch command to download the latest updates from the remote without merging or rebasing.

```
$ git fetch --all
```

Step 2: Use the git reset command to reset the master branch with updates that you fetched from remote. The hard option is used to forcefully change all the files in the local repository with a remote repository.

```
$ git reset hard <remote> / <branch_name>  
$ git reset-hard master
```

Git Pull Origin Master

We can pull the repository by using the Git pull command. The syntax is given below:

```
$ git pull <options> <remote> / <branch name>  
$ git pull origin master
```

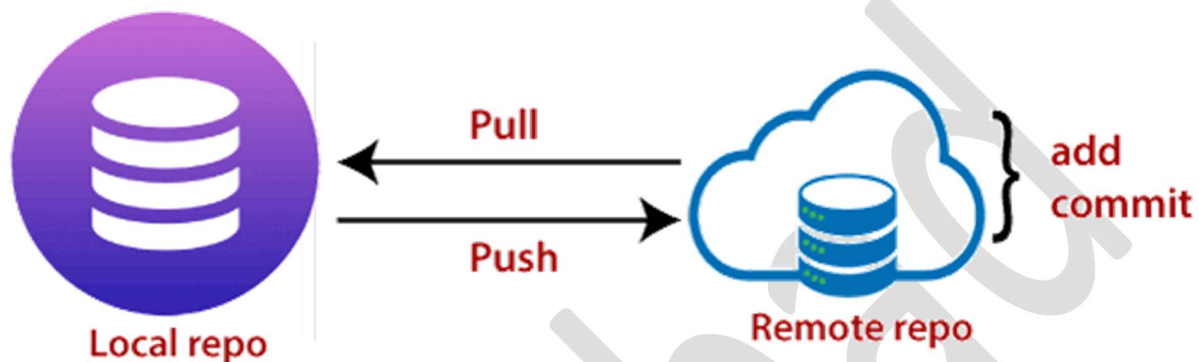
You can check the remote location of your repository. To do so type the below command:

```
$ git remote -v
```

Git Push

The Git push refers to upload local repository content to a remote repository.

The Git push command can be considered as a tool to transfer commits between local and remote repositories.



Syntax:

```
$ git push <option> [<Remote URL> <branch name> <refspec> ...]
```

Git Push Tags

<repository>: The repository is the destination of a push operation. It can be either a URL or the name of a remote repository.

<refspec>: It specifies the destination ref to update source object.

--all: The word "all" stands for all branches. It pushes all branches.

--prune: It removes the remote branches that do not have a local counterpart. Means, if you have a remote branch say demo, if this branch does not exist locally, then it will be removed.

--mirror: It is used to mirror the repository to the remote. Updated or Newly created local refs will be pushed to the remote end. It can be force updated on the remote end. The deleted refs will be removed from the remote end.

--dry-run: Dry run tests the commands. It does all this except originally update the repository.

--tags: It pushes all local tags.

--delete: It deletes the specified branch.

-u: It creates an upstream tracking connection. It is very useful if you are going to push the branch for the first time.

Git Push Origin Master

Generally, the term origin stands for the remote repository, and master is considered as the main branch. So, the entire statement "git push origin master" pushed the local content on the master branch of the remote location.

```
$ git push origin master
```

Git Force Push

The Git force push allows you to push local repository to remote without dealing with conflicts.

```
$ git push <remote> <branch> -f
```

Or

```
$ git push <remote> <branch> -force
```

How to safe force push repository:

```
$ git push <remote> <branch> --force-with-lease
```


Ashirbad