Git

Distributed Version Control System:

A distributed version control system (DVCS) brings a local copy of the complete repository to every team member's computer, so they can commit, branch, and merge and access files locally.

What is Git?

Git is a modern and widely used **distributed version control** system in the world. It is developed to manage projects with high speed and efficiency. The version control system allows us to monitor and work together with our team members at the same workspace.

Git is foundation of many services like **GitHub** and **GitLab**, but we can use Git without using any other Git services. Git can be used **privately** and **publicly**.

Git was created by **Linus Torvalds** in **2005** to develop Linux Kernel. It is also used as an important distributed version-control tool for **the DevOps**.

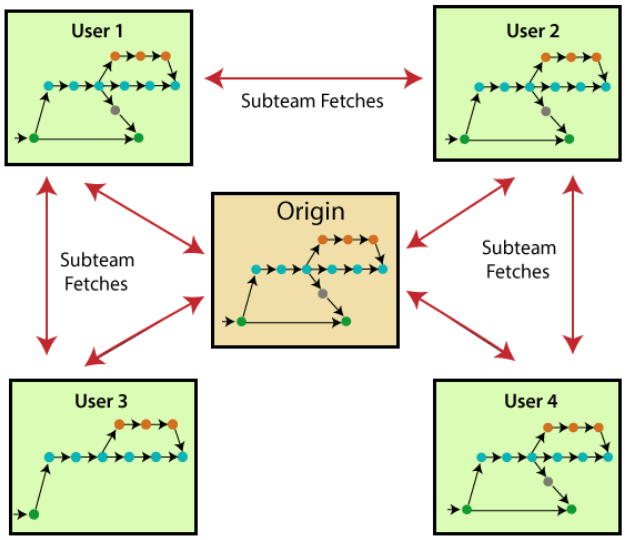
Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Features of Git:

- **Open Source**
  Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.

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

- **Distributed**
  One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.



- **Security:**
  Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

- **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. Also, a centralized version control system continually communicates with a server somewhere.

  Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages.

  Git was developed to work on the Linux kernel; therefore, it is capable enough to handle large repositories effectively. From the beginning, speed and performance have been Git's primary goals.

- **Supports non-linear development:**
  Git supports seamless branching and merging, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

- **Branching & Merging:**
  Branching and merging are the great features of Git, which makes it different from the other SCM tools. 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. Below are some features that can be achieved by branching:

  - We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.

- We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.

- We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.

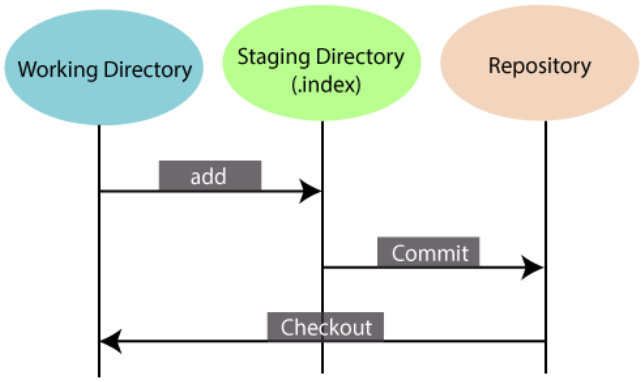
- The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

- **Data Assurances:**
  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. Most of the centralized version control systems do not provide such integrity by default.

- **Staging Area:**
  The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

  Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.



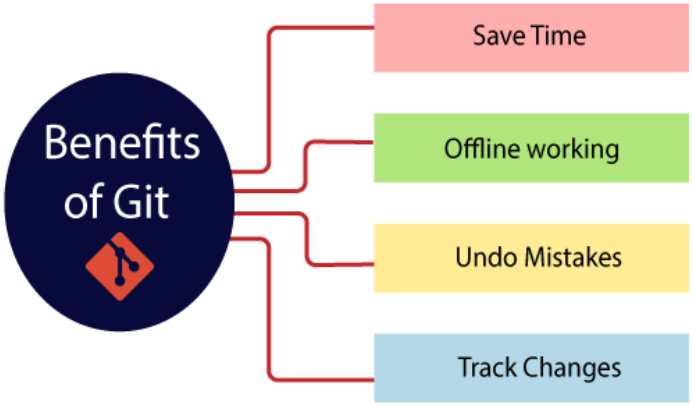
  Another feature of Git that makes it apart from other SCM tools is that it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.

- **Maintain the clean history:**
  Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

Benefits of Git

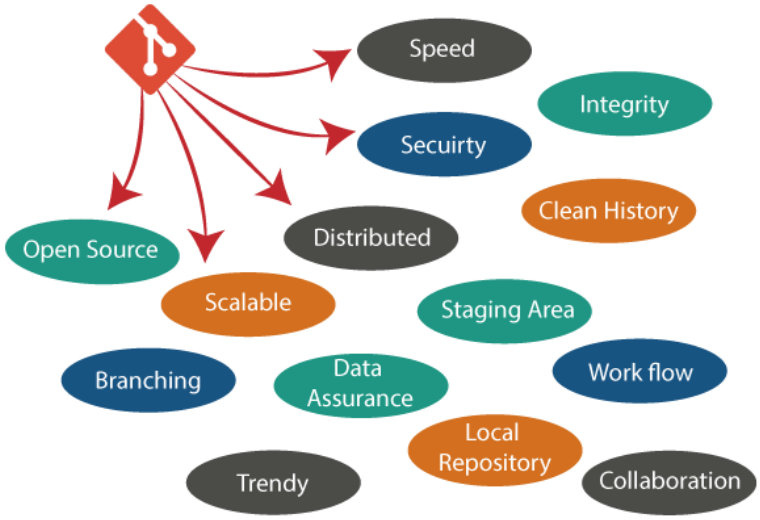
A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other

developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.



Why Git?

We have discussed many **features** and **benefits** of Git that demonstrate the undoubtedly Git as the **leading version control system**. Now, we will discuss some other points about why should we choose Git.



What is Git SSH Key?

An SSH key is an access credential for the SSH (secure shell) network protocol. This authenticated and encrypted secure network protocol is used for remote communication between machines on an unsecured open network. SSH is used for remote file transfer, network management, and remote operating system access. The SSH acronym is also used to describe a set of tools used to interact with the SSH protocol.

SSH uses a pair of keys to initiate a secure handshake between remote parties. The key pair contains a public and private key. The private vs public nomenclature can be confusing as they are both called keys. It is more helpful to think of the public key as a "lock" and the private key as the "key". You give the public 'lock' to remote parties to encrypt or 'lock' data. This data is then opened with the 'private' key which you hold in a secure place.

What is GitHub:

GitHub is a Git repository hosting service. GitHub also facilitates with many of its features, such as access control and collaboration. It provides a Web-based graphical interface.

GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.

It offers both **distributed version control and source code management (SCM)** functionality of Git. It also facilitates with 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. It hosts plenty of open-source projects and codes of various programming languages.

Some of its significant features are as follows.

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

Benefits of Git:

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.

- 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 versions.

Difference between Git & GitHub:

Git	GitHub
Git is software.	It is a service.
Linux maintains Git.	Microsoft maintains GitHub.
It is a command-line tool.	It is a graphical user interface.
You can install it locally on the system.	It is hosted on the web. It is exclusively cloud-based.
It focuses on code sharing and version control.	It focuses on centralized source code hosting.
It lacks a user management feature.	It has a built-in user management feature.
It is open-source licensed.	It has a free-tier and pay-for-use tier.

Git Environment Setup:

The environment of any tool consists of elements that support execution with software, hardware, and network configured. It includes operating system settings, hardware configuration, software configuration, test terminals, and other support to perform the operations. It is an essential aspect of any software.

It will help you to understand how to set up Git for first use on various platforms so you can read and write code in no time.

Git Commands:

Git Configuration:

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

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

The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.

- local
- global
- system

--local

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

--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:\Users\\.gitconfig"** on windows as a file format.

--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. It means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

- **Git config**
  Get and set configuration variables that control all facets of how Git looks and operates.

Set the name:
$ git config --global user.name "User name"

Set the email:
$ git config --global user.email "himanshudubey481@gmail.com"

Set the default editor:
$ git config --global core.editor Vim

Check the setting:
$ git config -list

- **Git alias**
  Set up an alias for each command:
  $ git config --global alias.co checkout
  $ git config --global alias.br branch
  $ git config --global alias.ci commit
  $ git config --global alias.st status

Stating a Project:

- **Git init**

  Create a local repository:
  $ git init
- **Git clone**

  Make a local copy of the server repository.
  $ git clone <link_gihub>

Local Changes:

- **Git add**

  Add a file to staging (Index) area:
  $ git add Filename

  Add all files of a repo to staging (Index) area:
  $ git add*

- **Git Commits**

  Record or snapshots the file permanently in the version history **with a message**.
  $ git commit -m " Commit Message"

  Delete particular commit using interactive rebase.
  $ git rebase -i HEAD~4 (using top 4 commit, delete/drop a particular commit. Use keyword: **drop** instead **pick;** [commit id will change])

Track Changes:

- **Git diff**

  Track the changes that have not been staged:
  $ git diff

  Track the changes that have staged but not committed:
  $ git diff –staged

  Track the changes after committing a file:
  $ git diff HEAD

  Track the changes between two commits:
  $ git diff Git Diff Branches:
  $ git diff < branch 2>

- **Git status**

  Display the state of the working directory and the staging area.
  $ git status

- **Git show Shows objects:**

  $ git show

Commit History:

- **Git log**

  Display the most recent commits and the status of the head:
  $ git log

  Display the output as one commit per line:
  $ git log –oneline

  Displays the files that have been modified:
  $ git log –stat

  Display the graphically:
  $ git log –oneline –graph –all
  $ git log –oneline –graph

  Display the modified files with location:
  $ git log -p

- **Git blame**

  Display the modification on each line of a file:
  $ git blame <file name>

Ignoring Files:

- **.gitignore**
  Specify intentionally untracked files that Git should ignore. Here in this file you can mention or add files which need to be ignore. Create ".gitignore" file:
  $ touch .gitignore

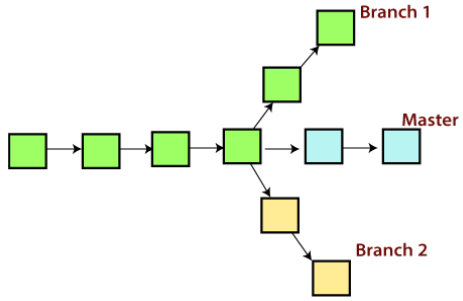
  List the ignored files:
  $ git ls-files -i --exclude-standard

Git Branching:

Git branch is a mechanism in the Git version control system that allows users to work on multiple versions of their codebase simultaneously.

A branch is essentially a separate line of development that can be used to experiment with new features or fixes without affecting the main codebase. Each branch is essentially a pointer to a particular commit in the Git repository, which allows users to switch between branches easily and quickly.

By using Git branches, developers can work on different features or bug fixes independently of each other, and then merge the changes back into the main codebase once they are complete and tested. This allows for greater flexibility and organization in the development process, and can help prevent conflicts between different sets of changes.



Master Branch:
The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

- **Git branch**

  Create branch:
  $ git branch <branch_name>

  List Branch:
  $ git branch --list

  Delete a Branch:
  $ git branch -d <branch_name>

Delete a remote Branch:
$ git push origin -delete <branch_name>

Rename Branch:
$ git branch -m <old_branch_name> <new_branch_name>

- **Git checkout**
  Switch between branches in a repository.

  Switch to a particular branch:
  $ git checkout <branch_name>

  Create a new branch and switch to it:
  $ git checkout -b <branch_name>

  Checkout a Remote branch:
  $ git checkout <branch_name>

- **Git stash**

  Switch branches without committing the current branch. Stash current work:
  $ git stash

  Saving stashes with a message:
  $ git stash save "messages"

  Check the stored stashes:
  $ git stash list

  Re-apply the changes that you just stashed:
  $ git stash apply

  Track the stashes and their changes:
  $ git stash show

  Re-apply the previous commits:
  $ git stash pop

  Delete a most recent stash from the queue:
  $ git stash drop

  Delete all the available stashes at once:
  $ git stash clear

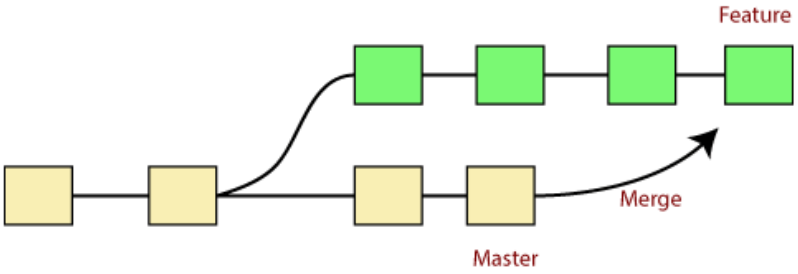
  Stash work on a separate branch:
  $ git stash branch

- **Git cherry pic**
  Apply the changes introduced by some existing commit:
  $ git cherry-pick

Git Merging:

Git merging is a process in the Git version control system that allows you to combine changes from different branches of your codebase. When you merge two branches, Git takes the changes from both branches and applies them to a new commit, creating a new branch that incorporates the changes from both branches.

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

Merging is an important tool in Git, as it allows developers to work on separate branches without interfering with each other's work. By merging changes from multiple branches into a single branch, you can keep your codebase up-to-date and ensure that all changes are properly incorporated into the final product.

- **Git merge**

  Merge the branches:
  $ git merge

  Merge the specified commit to currently active branch:
  $ git merge <commit_id>

  The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

  See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git add newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git commit -m "edited newfile1.txt"
[test d2bb07d] edited newfile1.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit d2bb07dc9352e194b13075dcfd28e4de802c070b (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 11:27:44 2019 +0530

    edited newfile1.txt

commit 2852e020909dfe705707695fd6d715cd723f9540 (test2, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added
```

Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout test2
Switched to branch 'test2'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge  d2bb07dc9352e194b13075dcfd28e4de802c070b
Updating 2852e02..d2bb07d
Fast-forward
 newfile1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$
```

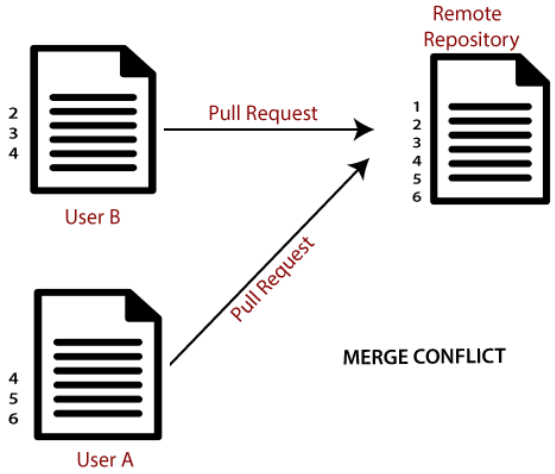
In the above output, we have merged the previous commit in the active branch test2.

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.

Git track each of lines. Merge Conflict happens when exact same line change in different branch.

For example:
In Master branch, we do some changes in a particular file and similarly we do some changes creating a new branch; after that we will try to combine, then conflict occurs.



❖ **Git rebase:**

Rebasing is the process of taking one branch and adding it to the tip of another, where the tip is simply the last commit in the branch.

Rebasing changes the history of entire branch.

Git rebase is a command in the Git version control system that allows you to modify the history of your Git repository. Specifically, Git rebase is used to change the base of a branch, which means that it allows you to move the starting point of a branch to a different commit.

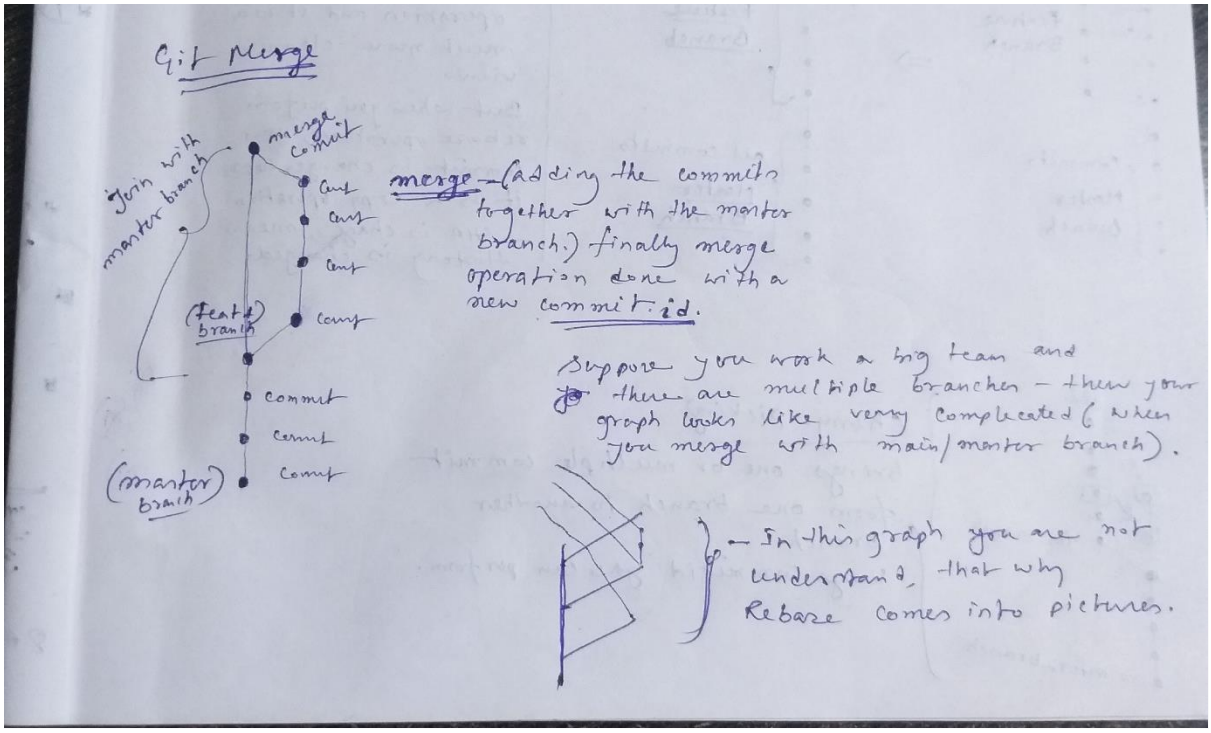
When you run the **git rebase** command, Git will take the commits in your current branch and apply them on top of a new base branch. This creates a new commit history that incorporates changes from both branches.

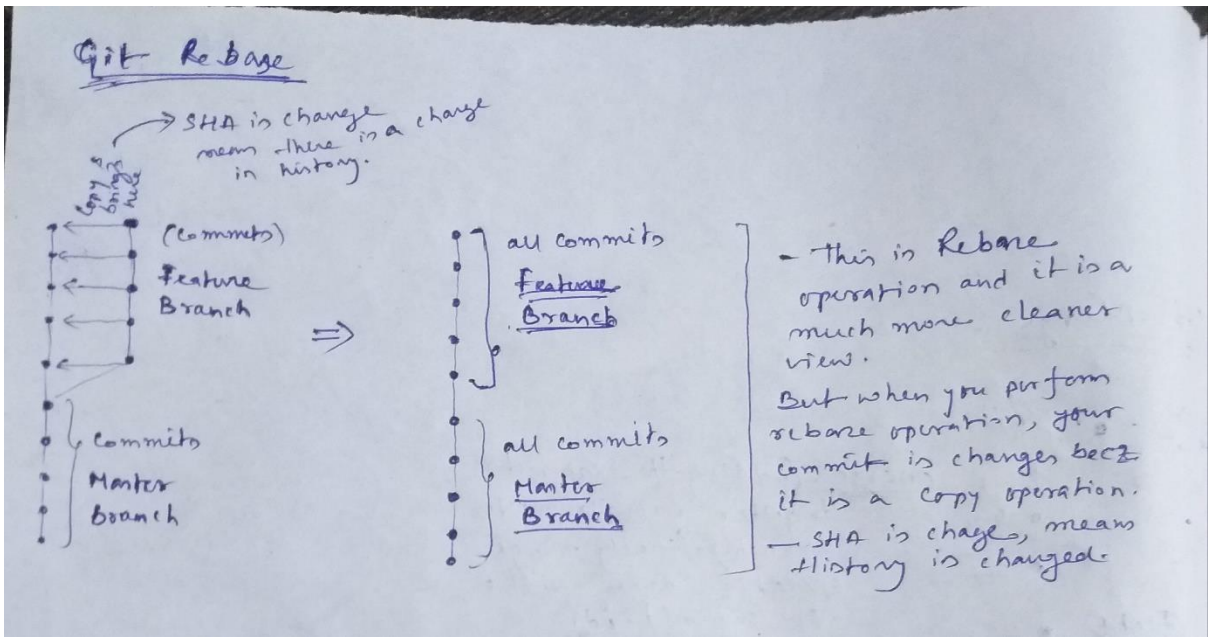
One of the main benefits of using Git rebase is that it can help you create a cleaner and more linear commit history. By combining multiple branches into a single branch, you can avoid creating unnecessary merge commits that clutter your commit history.

However, Git rebase can also be a potentially dangerous operation, especially if you're working with a public or shared branch. Since Git rebase modifies the history of a branch, it can lead to conflicts if other people are working on the same branch. In general, it's best to use Git rebase only for branches that you're working on alone.

If there are conflicts during a Git rebase operation, Git will prompt you to manually resolve them before continuing the rebase. Once the conflicts are resolved, Git will continue applying the remaining commits.

It's worth noting that Git rebase should be used with caution and only when it's appropriate. In general, it's best to use Git merge for branches that you're sharing with others, and Git rebase for branches that you're working on alone.





Apply a sequence of commits from distinct branches into a final commit.
$ git rebase

Continue the rebasing process:
$ git rebase -continue

Abort the rebasing process:
$ git rebase --skip

- **Git interactive rebase**
  Allow various operations like edit, rewrite, reorder, and more on existing commits. Interactive Rebasing changes the history of your branch. Different options are available:

$ git rebase -i

- **Squash:**
  To "squash" in Git means to combine multiple commits into one. You can do this at any point in time (by using Git's "Interactive Rebase" feature), though it is most often done when merging branches. Please note that there is no such thing as a stand-alone git squash command. Basically, squashing means melting the previous commits.
  Suppose there are 10 commits are there but you see that one commit is useful, then you merge all the commit into one.

  $ git rebase -i HEAD~4 (it means combine 4 commits into one commit. When you run this commit then code editor is opened and then use **squash** instead of **pick,** where **pick** where you want to squash).

```
pick 2db5bf1 Fruits Added1
squash 12a8d6b Fruits Added2
squash e9b8041 Fruits Added3
squash c4465be Fruits Added4

# Rebase fe0fdb1..c4465be onto fe0fdb1 (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
```

- **Amend:**
  The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. Basically, it helps you to do modification of last commit only.
  $ git commit –amend
  $ git commit –amend -m “messages or commits”
  $ git commit –amend --no-edit (you staged some files but not to commit or use previous commit)

- **Reward:**
  You can modify any commit using particular commit id (using 7-digit SHA)
  $ git rebase -i HEAD~2 (use keyword: **reward** instead **pick;** [commit id will change])

- **Delete:**
  Delete particular commit using interactive rebase.
  $ git rebase -i HEAD~4 (use keyword: **drop** instead **pick;** [commit id will change])

- **Reorder:**
  Basically, reorder the commit (means changing commit id position).
  $ git rebase -i HEAD~4 (re-order manually, just changing the position; here commit id will be change)

- **Split:**
  It means, splitting the particular commit
  $ git rebase -i HEAD~4 select particular commit which you want to split. Use **edit** instead of **pick**.
  Then new branch will be created and do split after that:
  $ git reset HEAD^
  check the git status:
  $ git status
  Staged the files:

$ git add <file_name_1> && $ git commit -m “message for file_1”
$ git add <file_name_2> && $ git commit -m “message for file_2”
Once you satisfied your changes then,
$ git rebase –continue
Check the commit status:
$ git log –oneline

Git Remote:

- **Git remote**
  Check the configuration of the remote server (Git remote supports a specific option -v to show the URLs that Git has stored as a short name. These short names are used during the reading and write operation. Here, -v stands for **verbose**):
  $ git remote -v

  Add a remote for the repository:
  $ git remote add <remote_name> <remote_url>

  Fetch the data from the remote server (You can fetch and pull data from the remote repository. The fetch and pull command go out to that remote server, and fetch all the data from that remote project that you don't have yet. These commands let us fetch the references to all the branches from that remote & The git pull command automatically fetches and then merges the remote data into your current branch. Pulling is an easier and comfortable workflow than fetching. Because the git clone command sets up your local master branch to track the remote master branch on the server you cloned.):
  $ git fetch <remote>
  $ git pull <remote>

  Remove a remote connection from the repository:
  $ git remote rm

  Rename remote server:
  $ git remote rename <old_remote_name> <new_remote_name>
  Show additional information about a particular remote (It will result in information about the remote server. It contains a list of branches related to the remote and also the endpoints attached for fetching and pushing):
  $ git remote show

  Change remote:
  $ git remote set-url <remote_name> <new_url>

- **Git origin master**
  Push data to the remote server:
  $ git push <short_name> <branch_name>
  Pull data from remote server:
  $ git pull origin master

Pushing Update:

- **Git push**

  Transfer the commits from your local repository to a remote server. Push data to the remote server:
  $ git push <remote_name> <branch_name>
  $ git push origin main

  Force push data:
  $ git push -f

  Delete a remote branch by push command:
  $ git push <branch_name> -delete
  $ git push origin -delete

Pulling Updates:

- **Git pull**
  git pull is actually a command that allows you to fetch from and integrate with another repository or local branch. A git pull is actually agit fetch followed by an additional action(s)- typically a git merge. (Update local from remote)

  Pull the data from the server:
  $ git pull origin master

  Pull a remote branch:
  $ git pull

- **Git fetch**
  Download branches and tags from one or more repositories. Basically, git fetch is a command that allow you to download objects from another repository, here after fetching you need to merging. (Update local from remote)

  Fetch the remote repository:
  $ git fetch < repository Url>

  Fetch a specific branch:
  $ git fetch

  Fetch all the branches simultaneously:
  $ git fetch -all

  Synchronize the local repository:
  $ git fetch origin

Undo Changes:

- **Git Revert:**

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

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Git revert undoes the changes made by a previous commit by creating an entirely new commits, all without altering the history of commits.

If a character is added in commit A, if git reverts commit A, then git will make a new commit where that character is deleted.

Undo the changes:
$ git revert

Revert a particular commit:
$ git revert <commit_id>

❖ **Git Reset:**

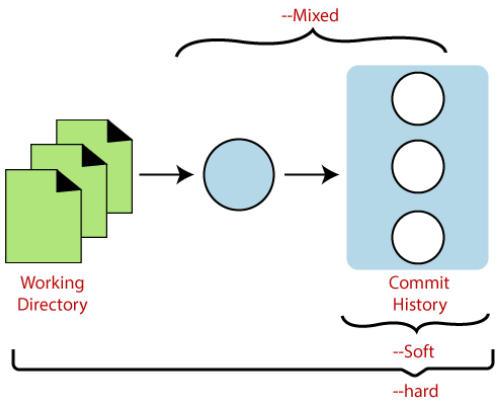
The term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.

- **Soft:**
  It is used to unstage the files which we have staged using the git add command.
- **Mixed:**
  It is used to remove the file which we have committed using the git commit command.
- **Hard:**
  It is used to remove all things which we have passed in our code.

If you do not mention anything then by default it is **Mixed.**

If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state. It is a sophisticated and versatile tool for undoing changes. It acts as a **time machine for Git**. You can jump up and forth between the various commits. Each of these reset variations affects specific trees that git uses to handle your file in its content.

Additionally, git reset can operate on whole commits objects or at an individual file level. Each of these reset variations affects specific trees that git uses to handle your file and its contents.



git reset --hard

git reset --hard is a command in the Git version control system that allows you to reset your working directory and index to a specific commit, discarding any changes that you have made since that commit.

When you run the **git reset --hard** command followed by a commit hash, Git will move the HEAD pointer and current branch pointer to the specified commit and reset your working directory and index to match that commit. This means that any changes that you have made since that commit will be lost, and your working directory will match the state of the repository at the specified commit.

It's worth noting that **git reset --hard** is a potentially dangerous command, as it can result in the loss of any changes that you have made since the specified commit. If you're not sure about the state of your repository or if you have uncommitted changes that you want to keep, it's best to use caution when running this command.

In general, **git reset --hard** should be used when you need to completely discard changes and start over from a specific point in your commit history. This command can be useful when you've made changes that you no longer need, or when you want to undo a series of commits and start over from an earlier point in your repository's history.

Generally, the reset hard mode performs 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 --soft

git reset --soft is a command in the Git version control system that allows you to reset your current branch pointer to a specific commit, without changing the state of your working directory or index.

When you run the **git reset --soft** command followed by a commit hash, Git will move the HEAD pointer and current branch pointer to the specified commit, but leave the state of your working directory and index unchanged. This means that any changes that you have made since the specified commit will still be present in your working directory and can be committed again.

The **git reset --soft** command can be useful when you want to undo a series of commits and start over from an earlier point in your repository's history, while keeping the changes that you've made since that point. For example, you can use **git reset --soft** to undo a series of commits that you've made on a feature branch and then merge those changes into a different branch.

It's worth noting that **git reset --soft** is a safer command than **git reset --hard**, as it allows you to undo commits without losing any changes that you've made since those commits. However, if you're not careful, **git reset --soft** can still cause problems, especially if you have uncommitted changes in your working directory or index. It's important to review the changes in your working directory and index carefully after running **git reset --soft** to ensure that everything is in the correct state before committing the changes again. (**$ git reset –soft HEAD~4** it means, 4 files are moved from staging area to unstaging area)

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.

git reset --mixed

git reset --mixed is a command in the Git version control system that allows you to reset your current branch pointer to a specific commit and reset the state of your index, but leave your working directory unchanged.

When you run the **git reset --mixed** command followed by a commit hash, Git will move the HEAD pointer and current branch pointer to the specified commit, and reset the state of your index to match that commit. However, the changes in your working directory will not be affected by this command. This means that any changes that you have made since the specified commit will still be present in your working directory and can be committed again after being added to the index.

The **git reset --mixed** command can be useful when you want to unstage changes that you've added to the index, or when you want to undo a series of commits and start over from an earlier point in your repository's history, while preserving the changes that you've made since that point.

It's worth noting that **git reset --mixed** is a potentially dangerous command, as it can result in the loss of changes in your index that have not been committed yet. If you're not sure about the state of your repository or if you have uncommitted changes that you want to keep, it's best to use caution when running this command.

In general, **git reset --mixed** should be used when you want to undo a series of commits and start over from an earlier point in your repository's history, while preserving the changes that you've made since that point, but without affecting the changes in your working directory.

Removing Files:

- **Git RM**

  git rm <file_name>

  git rm <file_names>

  In Git, 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. Additionally, it can be used to remove files from both the working directory and staging index.

  The files being removed must be ideal for the branch to remove. No updates to their contents can be staged in the index. Otherwise, the removing process can be complex, and sometimes it will not happen. But it can be done forcefully by **-f** option.

- **Git RM Cached**

  git rm --cached <file_name>

  git rm --cached <file_names>

  Sometimes you want to remove files from the Git but keep the files in your local repository. In other words, you do not want to share your file on Git. Git allows you to do so. The cached option is used in this case. It specifies that the removal operation will only act on the staging index, not on the repository.

Git Tagging:

Tagging is a mechanism used to create a snap shot of a Git Repository.

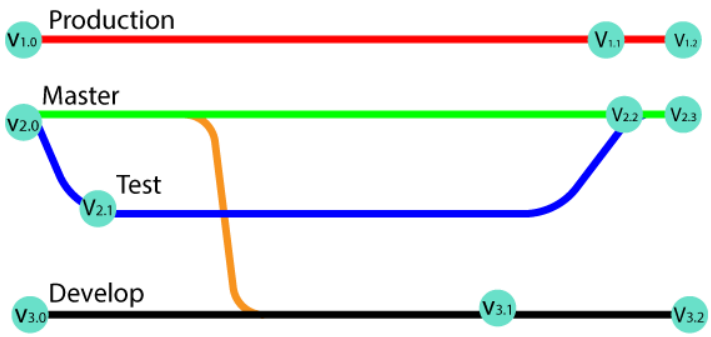
Tagging is traditionally used to create semantic version number identifier tags that corresponds to software release cycles.

Tags make a point as a specific point in Git history. Tags are used to mark a commit stage as relevant. We can tag a commit for 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. The below figure demonstrates the tags on various branches.

In the below image, there are many versions of a branch. All these versions are tags in the repository.

After multiple commits, we tag or create a version.



There are two types of tags:

- Annotated Tags
- Light-Weighted Tags

When to create tags:

- 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.

❖ **Annotated tag:**
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.

If you are pointing and saving a final version of any project, then it is recommended to create an annotated tag. But if you want to make a temporary mark point or don't want to share information, then you can create a light-weight tag. The data provided in annotated tags are essential for a public release of the project. There are more options available to annotate, like you can add a message for annotation of the project.
$ git tag -a **<tag** name**>**
$ git tag -a **<tag** name**>** <commit_id> (for specific commit, create a tag the you can use 7 digit SHA)
$ git tag **<tag** name**>** -m "**< Tag** message**>**

❖ **Light-Weighted Tags:**
Git supports one more type of tag; it is called as Light-weighted tag. The motive of both tags is the same as marking a point in the repository. 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, pass a tag name.
$ git tag **<tag** name**>**

- **List of Tags:**
  $ git tag
  $ git tag show <tag_name>
  $ git tag -l "**<pattern>***" (It displays the available tags using wild card pattern)

- **Delete a Tag:**
  $ git tag -d <tag_name>

Git FORKS:

A FORK is a copy of a particular repository. Forking a repository allows you to create freely experiments with changes without affecting the original projects. There is no such command for FORK. It means that you are doing copy operation of entire repository or mirroring operation of entire repo to my account.

It is used when:

- If I want to propose some changes to someone's project;
- Use existing project as a starting point;
- For bug fixing or resolving issue

After fixed the bug, then we create a pull request to the project owner. So that he/she can see and give some review the code and then attach in its base branch.
Forking is not a git function, it is a feature of git service like GitHub or Bitbucket, etc.

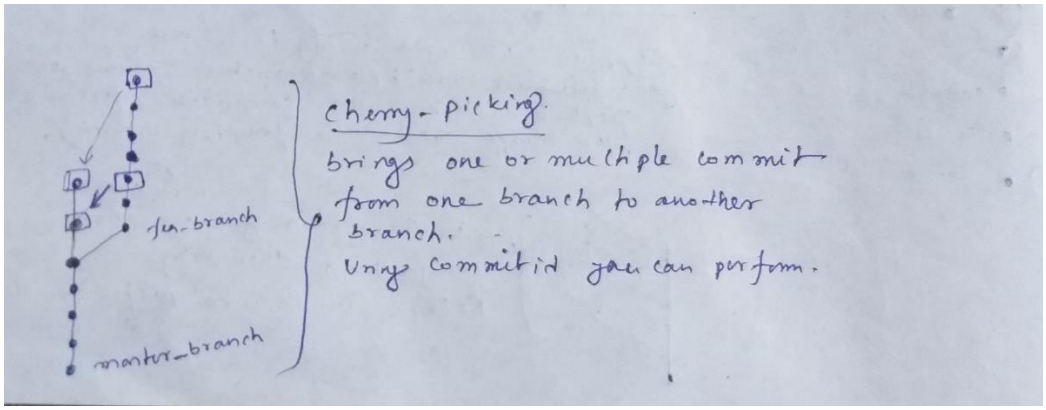
Reasons for Forking the repository:

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

Cherry Picking Git:

Cherry-picking in Git stands for applying some commit from one branch into another branch.
When we do cherry picking:

- Bring the changes from a specific commit.
- Choose one or commits.



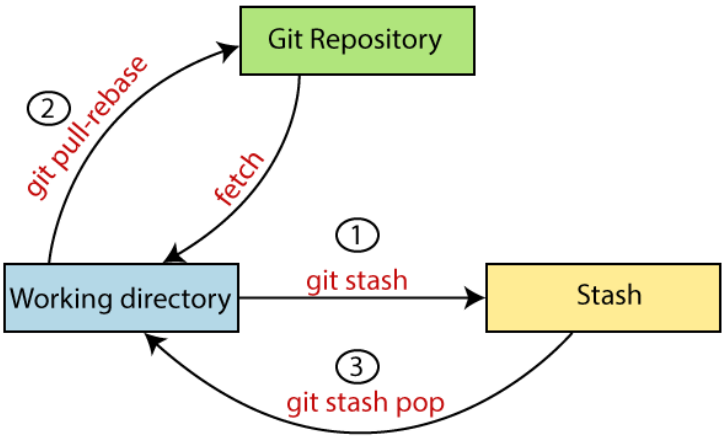
Switch to particular branch where you perform cherry-picking. Then file become available. And new commit id is generated.
$ git cherry-pick <commit_id>
You can perform multiple cherry-picking or bring the multiple commits one branch to another branch.
$ git cherry-pick <commit_id_1> <commit_id_2> <commit_id_3>
Suppose, if you want to bring the commit one branch to another branch and you don't want to commit, then you should use:
$ git cherry-pick <commit_id> -n
After that you can commit
$ git add <file_name> && $ git commit -m "messages"

Git Stashing:

Stashing saves a copy of your uncommitted changes i.e., working area and stashing area in a queue, off to the side of your project.

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.

The below figure demonstrates the properties and role of stashing concerning repository and working directory.



Generally, the stash's meaning is "**store something safely in a hidden place**." The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

$ git status
$ git stash
$ git status
$ git stash list

$ git stash -u (stash the untrack files. Move the untrack files from working directory to stashing directory)
$ git add <stash_file_name(s)>
$ git stash save "message" (save the stash with particular messages. If you do not mention or provide any message then it takes the first message where HEAD is pointing)

For view the stash (it showing the detailed overview of changes):
$ git stash show -p <stash_name> (stash_name came from stash list)

Move from Stash area to Working area, and then perform staging & commit:
$ git stash apply <stash_name> (it takes copy of files from stash area and then move to working area)
$ git add <file_name(s)>
$ git commit -m “messages”

$ git stash pop <stash_name> (it takes files from stash area and then move to staging area)
$ git commit -m “messages”

To clear entire stash:
$ git stash clear
$ git stash list (it shows empty list)

Create stash branch:
$ git stash branch <branch_name> <stash_name> (based on stash_name or a particular stash move to particular stash branch)
After that, you automatically switch to the particular branch that is created by stash branch.
$ git commit -m “messages”

Git REFLOG:

Reflog is a mechanism to record the changes in branches and saves the history (give detailed overview).
This command is to manage the information recorded in all branches.

Every action you perform inside of Git where data is stored, you can find it inside of the Reflog.
$ git reflog