# GIT

Git is a version control and source code management tool used to track changes to the source code and for rollback to previous versions etc. Git is an example of a **DVCS (Distributed Version Control System).** It is different from CVS (Centralized Version Control) or Subversion (also known as SVN). In Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

**Version Control:**

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control software keeps track of every modification to the code in a special kind of database.

* Multiple developers will work for project development
* Developers will be working from multiple locations
* All developers’ code should be stored at one place (Code Integration Should Happen)
* To integrate all the developers’ source code at one place we will use Source Code Repository Software

**Benefits of version control systems:**

* A complete long-term **change history** of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software.
* Creating a "**branch**" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict.

1) All the developers can connect to repository server and can integrate the code

2) Code Integration will become easy

3) Repository server will provide monitored access

- Who - When - Why- What

**Source code management:**

SCM tools provides a suite of helpful features to make collaborative code development a more user-friendly experience. This is an essential requirement for projects that involve multiple developers working on the same piece of application. Once SCM has started tracking all the changes to a project over time, a detailed historical record of the project’s life is created. This historical record can then be used to ‘undo’ changes to the codebase. The SCM can instantly revert the codebase back to a previous point in time.

Different vendors offer centralized code storage services integrated with git. Some of the popular services include:

* GitHub (Microsoft)
* Bitbucket (Atlassian)
* GitLab
* Code Commit (AWS)

**Environment Setup to work with Git Hub**

1) Create Github account ( www.github.com )

2) Download and install Git Client software ( https://git-scm.com/downloads )

(or) use command ‘sudo yum install git –y’

3) Once installation completed, right click on the mouse and verify git options display (If git options displaying our git client installation completed successfully)

**We can create 2 types of Repositories in Git Hub**

**1) Public repository:** Public Repository means everybody can access but we can choose who can modify our repository

**2) Private repository:** Private Repository means we will choose who can access and who can modify

**Setting up a repository:**

If setting up a repository from your local project folder, we need to first cd into the root directory of the project and then execute **‘git init’** command. Or else we can simply pass the directory as an argument to the command without having to move to the directory.

* **git init <project\_directory>**

More often, the project will be initialized in a central repository and we have to copy it into our local environment. In such cases we use the following command.

* **git clone <repository\_url>**

This creates a copy of the existing central repository in our local system and we can work on it.

**Configuration & set up: git config:**

By cloning a repository, a connection between the local and central repositories will be established by default. In case of initiating the local repository exclusively we need to add the link to central repository using the following command.

* **git remote add origin <repository\_url>**

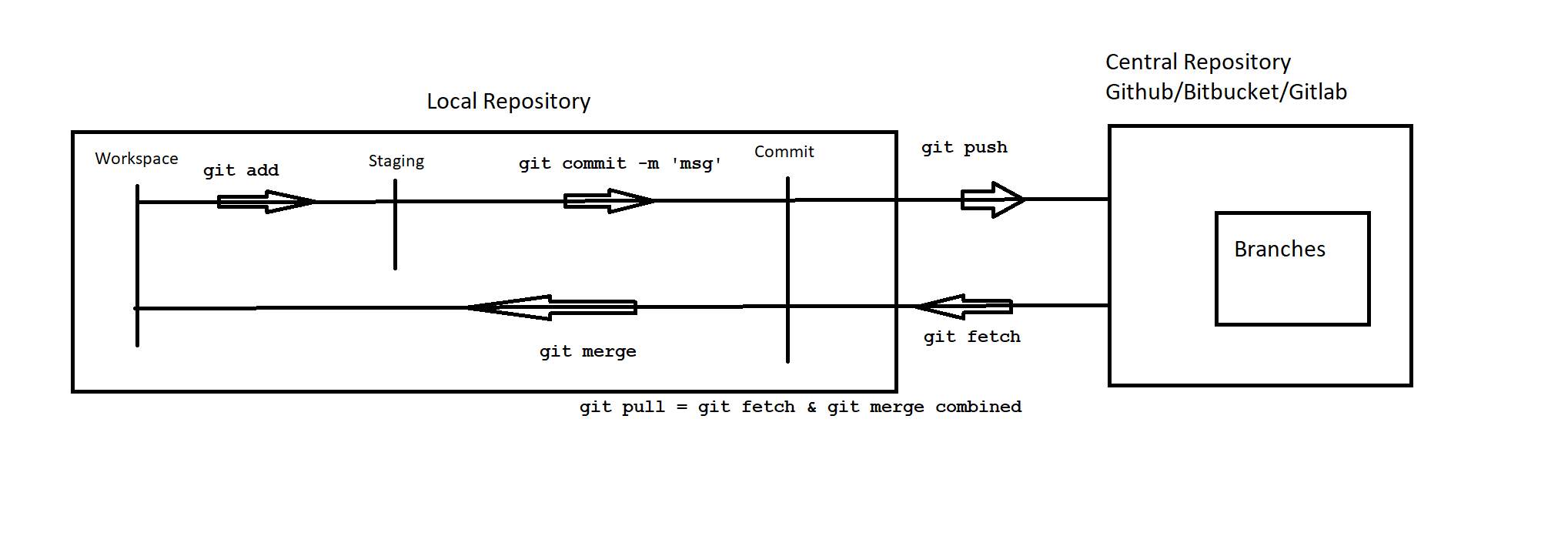
Once we have mapped the remote repository, we can push our local branch changes to the remote repository using the following command.

* **git push -u origin <local\_branch\_name>**

To define the author name to be used for all commits in the current repository. Typically, you’ll want to use the **‘—global’** flag to set configuration options for the current user.

* **git config --global user.name <name>**
* **git config --global user.email <email>**
* **git config --global credential.helper store**

Representation of Git workflow



**Common Steps in Git workflow:**

1. Git repository is initialized at an existing project directory or a new empty directory.

**git init**

1. Alternatively, an existing remote repository can be cloned.

**git clone <remote> <url>**

1. The changes are made to the project in working space and they are then marked for staging.

**git add <filename> (or) git add .**

1. All the staged changes are reviewed and a commit snapshot is created on the local repository.

**git commit -m “message”**

1. The commits made on the local repository can be pushed to remote repository.

**git push <remote> <branch>**

Apart from the above the other commands used for various operations in a git repository are listed below.

**Git commands Cheat Sheet**

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| git config --global user.name <name> | Define the author name to be used for all commits by the current user. | git config --global user.name “Raghu ” |
| git config --global user.email <email> | Define the author email to be used for all commits by the current user. | git config --global user.email “javabyraghu@email.com” |
| git config credential.helper store | Store password in configurations file | git config credential.helper store |
| git init (or)  git init <dir\_name> | Initialize the current or an existing directory as a Git repository | git init “~/BashScripts” |
| git clone <url> | Clone an existing repository from a remote location url | git clone “https://github.com/xyz/BashScripts.git” |
| git status | Show modified files both in working directory and also staged for your next commit | git status |
| git add <filename> (or) git add . | Mark a modified file or all modifications for staging before commit | git add samplefile.txt |
| git reset [file] (or)  git reset | Unstage a given file or all the files in staging | git reset filename |
| git diff | Shows the difference between the working directory and staging | git diff (or)  git diff <filename> (or)  git diff <commit1> <commit2> |
| git commit -m “message” | Commit your staged content as new commit snapshot | git commit -m “this is a commit” |
| git commit --amend “message” | Used to edit the commit message of last commit and add new changes to the existing previous commit. All changes to working directory will be added to the last commit and working directory will be set clean | git commit --amend “The last commit message is now edited” |
| git stash | Cleans the working directory and staging and creates a stash of present changes to work on them later | git stash |
| git stash push -m “message” | It creates stash with our message for identification | git stash push -m “added new lines in index.html” |
| git stash list | Shows the list of stashed changes with latest on top | git stash list |
| git stash apply | Brings the changes in latest stash to working directory | git stash apply |
| git stash pop | The latest stash is applied and dropped or index of particular stash may be specified | git stash pop 1 |
| git stash drop | Deletes the specified stash entry or the latest stash entry | git stash drop 2 |
| git clean | Cleans and deletes untracked files in the repository | git clean |
| git reset --hard <commit> | Clears the changes in working directory and staging to the specified commit. Deletes all commits after the specified commit | git reset --hard 7c44543 |
| git branch | Lists the current branches in the repository with \* marked against current branch | git branch |
| git branch <branch\_name> | Creates a new branch with the specified branch name | git branch NewFeature |
| git checkout | Switches to other branch or commit id as specified | git checkout feature\_branch |
| git checkout -b <branch\_name> | Creates a new branch and switches head to the new branch | git checkout -b new\_feature |
| git merge <branch\_name> | Merges the specified branch to the current HEAD branch | git merge feature1 |
| git rebase <branch\_name> | Reset the base of current branch with respect to the upstream branch to the latest commit of the upstream branch | git rebase feature1 |
| git revert <commit> | Creates a new commit that undoes all the changes made in the specified commit. | git revert 7c44543 |
| git remote add <name> <url> | Creates new connection for a remote repository. After creating a connection <name> can be used as an alias for <url> | git remote add origin “https://github.com/xyz/repository.git” |
| git fetch <remote> <branch> | Fetches a specific <branch>, from the repo. Leave off <branch>  to fetch all remote refs. | git fetch origin feature1 |
| git pull <remote> | Fetch the remote repositories copy of current branch and immediately merge it into current branch | git pull origin |
| git push <remote> <branch> | Push the branch to remote repository along with all commits. If there is not branch in remote a new branch will be created. | git push origin feature1 |
| git log (or)  git log -limit | Shows the details of all commits or specified number of commits | git log -5 |
| git log --oneline | Displays the details of all commits briefly with oneline for each entry | git log –oneline -5 |
| git log --graph | Shows a graphical representation of the flow of commits. | git log –graph |
| git log --all | This includes all the commits across all branches in the repository. | git log --oneline --all |

### BRANCH COMMANDS:

* Run this command:
  + **git status**

### List All Branches

**NOTE:**The current local branch will be marked with an asterisk (\*).

* To see **local branches**, run this command:
  + **git branch**
* To see **remote branches**, run this command:
  + **git branch -r**
* To see **all local and remote branches**, run this command:
  + **git branch -a**

### Create a New Branch:

* Run this command (replacing **my-branch-name**with whatever name you want):
  + **git checkout -b my-branch-name**
* You're now ready to commit to this branch.

### Switch to a Branch in Your Local Repo

* Run this command:
  + **git checkout my-branch-name**

### Switch to a Branch That Came From a Remote Repo

* To get a list of all branches from the remote, run this command:
  + **git pull**
* Run this command to switch to the branch:
  + **git checkout --track origin/my-branch-name**

### Push to a Branch

* If your local branch **does not exist**on the remote, run either of these commands:
  + **git push -u origin my-branch-name**
  + **git push -u origin HEAD**

**NOTE:**HEAD is a reference to the top of the current branch, so it's an easy way to push to a branch of the same name on the remote. This saves you from having to type out the exact name of the branch!

* If your local branch **already exists**on the remote, run this command:
  + **git push**

### Merge a Branch

* You'll want to make sure your working tree is clean and see what branch you're on. Run this command:
  + **git status**
* First, you must check out the branch that you want to merge another branch into (changes will be merged into this branch). If you're not already on the desired branch, run this command:
  + **git checkout master**

**NOTE:** Replace **master**with another branch name as needed.

* Now you can merge another branch into the current branch. Run this command:
  + **git merge my-branch-name**

### Delete Branches

* To delete a **remote branch**, run this command:
  + **git push origin --delete my-branch-name**
* To delete a **local branch**, run either of these commands:
  + **git branch -d my-branch-name**
  + **git branch -D my-branch-name**

**NOTE:** The -d option only deletes the branch if it has already been merged. The -D option is a shortcut for --delete --force, which deletes the branch irrespective of its merged status.

## FAQs:

**Q) What is git reflog?**

A) ‘git reflog’ gives the complete history of updates on all branches including branch switch, pulling of changes, merge, local modifications, new commits and also resets. This is beneficial in case of hard reset and the commit history is lost in the HEAD git log. Git reflog also contains the data for hard reset commits.

**Q) What is git rebase?**

A) ‘git rebase’ is used to reset the base commit for the present branch from the upstream or parent branch. This base can be set to the latest commit in the upstream branch, a tag or a specific commit id.

Using git rebase all the commits in the parent branch are added and the commits of the existing branch will be shifted to the end with a new commit hash.

**Q) How to revert the last commit in git?**

A) git revert <commitid>

**Q) What is git diff?**

A) git diff is used to compare the changes in a specific file or a group of files between the present version of the file and its state on the provided commit id.

This command also gives the changes that are made to the files since last commit.

**Q) Explain the command – ‘git log --graph --oneline --decorate’**

A) git log --graph is used to display the log in an interactive graphical format

**Q) How to add a collaborator in GitHub and restrict permissions with example?**

A)

* Open the current repository in GitHub.
* Navigate to the settings tab.
* Click on Collaborators on the left-hand side menu.
* Click on ‘Add people’ button.
* You can add user by specifying their GitHub username or email id or full name.
* An invitation will be sent to the specific user to record their consent.
* After the user accepts the invitation they are assigned as a collaborator.
* The collaborator can now work on unprotected branches in the repository and commit their changes.
* To restrict the collaborator from making changes to a specific branch (like main/master), mark the branch as protected.
* Go to settings and select ‘Branches’ from the left-hand side menu.
* Give the branch name and select the appropriate protection required by you.

**Q) What is git fork? What is the difference between fork, branch and clone?**

A) git fork on GitHub is a convenient way of copying a specific repository to a GitHub user’s local account.

**Q) What is the difference between git pull and git fetch?**

A) git fetch is used to update the local copy of remote repository (popularly called origin) with the latest changes applied to the remote repository. It doesn’t merge the changes to the local repository.

However, git pull updates the local copy of the remote repository and merges the changes to the local repository.

**Q) Explain the difference between HEAD, working tree and index in git?**

A) HEAD is a symbolic reference, it is pointing by default to the last commit in the currently active branch. HEAD can also be used as a reference variable for currently active branch.

Working Tree in Git is a directory (and its files and subdirectories) on your file system that is associated with a repository.

Git index is a critical data structure in Git. It serves as the “staging area” between the files you have on your file system and your commit history. When you run git add, the files from your working directory are hashed and stored as objects in the index, leading them to be “staged changes”

**Q) Name some commonly used branches in GitHub – master/main, stage, hotfixes, feature.**

A) **master/main** – Default branch on a newly created git repository. Usually, the Production ready application code of the system is stored in master/main branch and it reflects the code base of presently running version of the application. No commits are directly made to the main/master branch and it should be accepting pull requests from other branches after proper review and approval from the product owner/client.

**staging branch** – The staging environment is as similar to production as possible. The idea is that if a feature is running perfectly fine in staging, it will run in production also. Smoke tests and most of the performance, reliability, security and availability testing will be carried out in this environment. The customers/clients will be shown a demo in this environment before releasing it to production and making the application live.

**hotfix branch** – This branch is created when there is an urgent to release a patch to the production environment. It may be to fix a bug or a critical security flaw.

**feature branch** – This branch is created for each new feature that is being developed. Direct commits can be made if an individual developer is working on the feature. If a team of developers is working, the feature will be owned by one of the developer and pull requests are created from the individual user branches to the feature branch.

**Q) What is the overall flow of code in git branches?**

The overall flow of Git is as follows:

* A ***develop*** branch is created from ***main***
* A ***release*** branch is created from ***develop***
* ***Feature*** branches are created from ***develop***
* When a ***feature*** is complete it is merged into the ***develop*** branch
* When the ***release*** branch is done it is merged into ***develop*** and ***main***
* If an issue in ***main*** is detected a ***hotfix*** branch is created from ***main***
* Once the ***hotfix*** is complete it is merged to both ***develop*** and ***main***

**Q) What is .gitignore?**

A) .gitignore is a file in the git repository which contains the list of files that should be excluded from the index of the git repository. The ignored files could contain sensitive data such as passphrases, access keys or some random log files. The use case for .gitignore varies based on the project.

**Q) What is stash and pop?**

A) git stash is used as a temporary storage for workspace data. The data can be retrieved later to the HEAD branch or to a new branch. Normally applying a stash doesn’t delete the stash from the stash list. If we need to apply and delete the latest stash data, we need to use the command ‘git stash pop’.