GIT

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

## Version Controlling

This is the process of preserving older and latest versions of the code. All the developers upload their code into the VCS. The VCS's accept the code uploads coming from various developers and create one integrated project out of those uploads. The next time when developers download the code from the VCS they will find the code created by the entire team. Process of uploading into VCS is called "check in" and downloading from VCS is called "checkout"

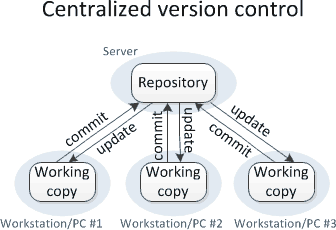
VCS's also preserve the different versions of the code so that the developers can switch between any version based on requirement. VCS's also track who is making what kind of changes VCS's are of two types

1. Centralized Version Controlling
2. Distributred Version Controlling

### Centralised Version Controlling

Here we have a remote server where version controlling happens.All the developers checkin their code into this central server.

Eg: SVN(SubVersion)



### Distributred Version Controlling

Here a local repository is installed on every developers machine where version controlling happens at the level of the individual developer.Later from the LR the code is uploaded to the remote repository where version controlling happens at the level of the complete team.

## Installation and configuration

### Installing git on windows

1. Open https://git-scm.com/downloads
2. Download git for windows--->Install it
3. Open git bash
4. Execute the git commands

### Installing git on a Linux(Ubuntu)

1. Update the apt-repository sudo apt-get update
2. Install git

sudo apt-get install -y git To check the version of git git –verion

### Configuration:

Setting username and email globally for all users git config --global user.name "user name"

git config --global user.email "[user@gmail.com](mailto:user@gmail.com)"

To see the list of all default configuration git config --global --list

## Local Repo:

Git when working on the local machine uses three sections.

1. Working directory (workspace)
2. Stagging Area
3. Local Repository

### Working directory

Initially all the files created by the developers are stored in a folder called working directory and these files are initially called untracked files.

### Stagging Area

Files from working directory are moved to an intermediate memory area called stagging area. The files present here are called stagged files.

### Local Repository

All files from stagging area will be moved into the local repository and this is where version controlling happens.These files are called commited files

## Basic commands

1. To initilise the working dir to accept git commands. Open git bash cd path\_of\_working\_dir

git init

The above command will create a hidden folder called .git

where it stores some configuration files for maintianing the local repository

1. To send file from working dir to stagging area git add filename

To send multiple files into stagging area git add file1 file2 file3

To send all files and fodlers into the stagging area git add .

1. To unstage files ie bring files back from stagging to untracked section git rm --cached filename

(or)

git reset filename

1. To send files from stagging to local repository git commit -m "Some commit message"
2. To see the status of the untracked and stagging section git status
3. To see the total commits that are done in LR git log

To see the commit history in a simplified format git log --oneline

### Git log and git reflog:

git log shows the current HEAD and its ancestry. That is, it prints the commit HEAD points to, then its parent, its parent, and so on. It traverses back through the repo's ancestry, by recursively looking up each commit's parent.

*(*In practice, some commits have more than one parent. To see a more representative log, use a command like git log --oneline --graph --decorate.)

git reflog doesn't traverse HEAD's ancestry at all. The reflog is an ordered list of the commits that HEAD has pointed to: it's undo history for your repo. The reflog isn't part of the repo itself (it's stored separately to the commits themselves) and isn't included in pushes, fetches or clones; it's purely local.

Aside: understanding the reflog means you can't really lose data from your repo once it's been committed. If you accidentally reset to an older commit, or rebase wrongly, or any other operation that visually "removes" commits, you can use the reflog to see where you were before and *git reset --hard* back to that ref to restore your previous state. Remember, refs imply not just the commit but the entire history behind it.

## .gitignore

This is a special file where we can store the private filenames. Any filename that is mentioned in .gitignore will not longer be accessed by git

1. Create few files

touch file1 file2 file3 file4

1. Check the status of git git status

It will show all the above 4 files as untracked file

1. Create a hidden file .gitignore and enter the above filenames cat > .gitignore

file1 file2 file3 file4

To come out of cat command press ctrl+d (EOF)

1. Check the status of git git status

It will not longer show the above created four files as untracked

## Git Branching

This is a feature of git using which the developers can create separate functionalities of code on different branches and later merge them with the master branch. This will help in creating the code in an uncluttered way

1. To see the list of all branches git branch

To see all branches (local and remote) git branch -a

1. To create a branch git branch branchame
2. To move into a branch git checkout branchname
3. To create a branch and also move into it git checkout -b branchname
4. To merge a brnach with master First move to master

git checkout master git merge branchname

1. To delete a branch that is merged git branch -d branchname

This is also called soft delete

1. To delete a branch that is not merged git branch -D branchname

This is also called hard delete

Note: whenever a branch is created whatever is the commit history of the parent branch will be copied to the newly created branch

Note: Irrespective of where a file is created or modified that file belongs to that branch form where it is commited

## Working on Git remote repository ("github")

1. Open github.com
2. Signup for a free account
3. Signin into that account

### To push the code from local repository to remote repository

1. Signin into github.com
2. Click on + on top right corner
3. click on New repository
4. give some name for repository
5. Select public--->Click on Create repository
6. Go to "Push an exisiting repository from command line"
7. Copy the 1st command and paste in gitbash

This will create a link between the local repository and the remote repository

1. Copy the second command and paste in gitbash

Enter username and password of github this process is called checkin

### Downloading from remote git repository

This can be done in 3 ways 1 git clone

1. git fetch
2. git pull

### git clone

This will download the entire remote repository from git hub into individual developers machine irresepective of whether that code is already present or not

Syntax

git clone remote\_repo\_url

### git fetch

This will work only when there are modifications in the code between the local repository and remote repository

git fetch will download the modified files and place them on a remote branch.We can go to that remote branch check if those changes are accptable or not and if they are acceptable we can merge them with master

1. Create some modifications to any file on github Open github.com

Click on our remote repository

click on any file that we want to modify Click on Edit icon

Make some changes--->click on commit changes

1. In git bash git fetch
2. Check the modified file on master branch We will not see any modifications
3. The modifications will be on the remote branch Move to the remote branch

git branch -a

git checkout -b remotes/origin/master

1. View the modified file

If the changes are accpetable merge with master git checkout master

git merge remotes/origin/master

### git pull:

This will also work only when there are modifications between the local repository and the remote repository. But it will directly merge the modified files with master branch

1. Open github.com
2. Make some changes to a file--->commit changes
3. In git bash git pull

The modified files can be directly seen on the master branch

## Git merge and git rebase

### git merge:

In git merge the commits are inserted into the branch that we are merging exactly at the location that they were created. Which means the commits coming from a branch can look like older commits on the master

1. On the master create few commits touch f1

git add .

git commit -m "a" touch f2

git add .

git commit -m "b"

1. Create a new branch and create few commits on it

git checkout -b sprint-1 touch f3

git add .

git commit -m "c" touch f4

git add .

git commit -m "d"

1. Move back to master and create few more commits git checkout master

touch f5 git add .

git commit -m "e" touch f6

git add .

git commit -m "f"

1. Merge sprint-1 with master git merge sprint1
2. Delete sprint-1 branch git branch -d sprint-1
3. Check commit history of master git log –oneline

### git rebase:

This is used for performing a fastforward merge ie the commits coming from the branch are added to the top of the master branch.This is helpful when we want the commits coming from a branch to be reflected as the latest working version of code on master.

1. On the master create few commits touch f1

git add .

git commit -m "a" touch f2

git add .

git commit -m "b"

1. Create a new branch and create few commits on it git checkout -b sprint-1

touch f3 git add .

git commit -m "c" touch f4

git add .

git commit -m "d"

1. Move back to master and create few more commits git checkout master

touch f5 git add .

git commit -m "e" touch f6

git add .

git commit -m "f"

1. Rebase sprint-1 with master

git checkout sprint-1 git rebase master

git checkout master git merge sprint-1

## Rearranging the commit history

The commit history can be changed according to our requirment using the git rebase command. The latest commit or the top most commit is genrally called HEAD.We can make this HEAD point to any older commit and make that older commit as the latest commit

Note: The very first commit is called as initial commit and this cannot be rearranged git rebase -i HEAD~4

The number 4 represents the top 4 comits that we want to rearrange

The above command will open the commit hsitory in vi editor where can can simply change the commit order Save and Quit Esc :wq Enter

## Git Squash

Squash is merging of commits to make multiple commits to make it look like a single commit. git rebase -i HEAD~4

The above command will open in vi editor where the complete commit history is shown

Remove the pick word and replace it with squash Save and quit esc : wq Enter

Check the commit history git log --oneline

## Modifing exisiting commits

Whenever we modify a file or create new files generally we create a new commit.Instead we can put the modifications in the existing commit itself rather than creating a new commit.

This can be done using git **amend** command

1. Create few commits in git LR touch f1

git add .

git commit -m "a" touch f2

git add .

git commit -m "b" touch f3

git add .

git commit -m "c"

1. Check the commit history git log --oneline
2. Modify some files or create new files cat > f3

Put some data

ctrl+d to comeout of cat

1. check status of git git status

We will find a modified file

1. Send it to stagging area

git add .

1. Commit it to local repositry as an exisiting commmit git commit --amend -m "c"
2. Check the commit history git log --oneline

We will see that new commit is not created the changes are pushed into the exisiting "c" commit

Note: git amend actually creates a new commit.

The older "c" commit becomes an orphaned commit which Can not be see using git log

We can use "git reflog" for seeing the entire commit history.ie active and orphaned commits

## Git cherry pick

This is used for choosing which commits we want to take into the master branch Generally when we perform "git merge" or "git rebase" all the commits of that branch will come into master branch

Cherry pick will allow us to select only those commits that we require and merge them with master

1. Create few commits on master touch f1

git add .

git commit -m "a" touch f2

git add .

git commit -m "b"

1. Create a branch and some commits on it

git checkout -b sprint-1

1. Create some commits here touch f3

git add .

git commit -m "c touch f4

git add .

git commit -m "d" touch f5

git add .

git commit -m "e"

1. Check the commit history on sprint-1 branch git log --oneline
2. Identify the commits that we want from sprint-1 branch and move back to master branch git checkout master

git cherry-pick commit\_id1 commit\_id2

1. Check the commit history on master git log --oneline

## git stash

This is a feature of git which is used for leaving unfinished work and start a new functionality related coding. Further command nof git should not touch the unfinished files

1. To stash the files present in stagging area git stash
2. To stash the files present in stagging area and untracked section git stash -u
3. To stash the files present in stagging area, untracked section and also the .gitignore git stash -a
4. To see the list of all stashes that we have done git stash list
5. To unstash the latest stash git stash pop
6. To unstash any older stash git stash pop stash{stashno}

Eg: To unstash the second last stash git stash pop stash{1}

### Next case on stash

1. Create a file and send to stagging area touch f1

git add .

1. Stash the above file git stash
2. Create a new file and stash it touch f2

git stash -u

1. Create few new files and place them in .gitignore Stash .gitignore also

touch f3 f4 f5

cat > .gitignore f3

f4 f5

Press Ctrl+d to come out of cat command

1. Since f3, f4, f5 are put in .gitignore git status will no longer

show them as untracked files but it will show .gitignore as untracked file.ie further commands of git will send this .gitignore into stagging area and also in local and remote repositories.If we want ot avoid that and

.gitignore should not be accessed by git git stash –a

## git squash:

One very nice feature of Git is the ability to rewrite the history of commit.

The principal reason for doing this is that a lot of such history is relevant only for the developer who generated it, so it must be simplified, or made more nice, before submitting it to a shared repository.

**Squashing a commit** means, from an idiomatic point of view, to move the changes introduced in said commit into its parent so that you end with one commit out of twos.

If you repeat this process multiple times, you can reduce *n* commit to a single one.

Use git rebase -i <after-this-commit> and replace "pick" on the second and subsequent commits with "squash" or "fixup", as described in [the manual.](http://git-scm.com/docs/git-rebase#_interactive_mode)

In this example, <after-this-commit> is either the SHA1 hash or the relative location from the HEAD of the current branch from which commits are analyzed for the rebase command. For example, if the user wishes to view 5 commits from the current HEAD in the past the command is git rebase -i HEAD~5

**Tagging in Git**

Tags are used for placing bookmarks on commits. They are to specify info related to who tagged , when it was tagged and why it was tagged. Generally used for releases. This helps in understanding what are the commits that are related to specific releases

Tags are of two types

1. Light weight tags
2. Annoted tags

### Light weight tags

1. To create a light weight tag git tag tabname
2. To see the list of all the tags git tag
3. To create an annoted tag

git tag -a tagname -m "some message"

**Note**: Tags are always created for the latest commit (HEAD)

1. To create tags for an older commit

git tag -a tagname -m "message" commit\_id

1. To delete a localtag git tag -d tagname
2. To push all tags into github git push --tags
3. To delete tags from the remote github git push origin :tagname

## git diff

This is used for finding the difference between 2 commits or it can be used for finding the difference between a commit and a file yet to be commited

1. To find diff between 2 commits git diff commit1\_id commit2\_id
2. To find diff between latest commit and a file git diff HEAD filename

## git revert:

Given one or more existing commits, revert the changes that the related patches introduce, and record some new

commits that record them. This requires your working tree to be clean (no modifications from the HEAD commit).

Note: *git revert* is used to record some new commits to reverse the effect of some earlier commits (often only a faulty one). If you want to throw away all uncommitted changes in your working directory, you should see [git-](https://git-scm.com/docs/git-reset) [reset[1]](https://git-scm.com/docs/git-reset), particularly the --hard option. If you want to extract specific files as they were in another commit, you should see [git-checkout[1]](https://git-scm.com/docs/git-checkout), specifically the git checkout <commit> -- <filename> syntax. Take care with these alternatives as both will discard uncommitted changes in your working directory.

### usage

Git revert <commit-id>