Git Document

Ashoka Chakravarti Marpu

http://www.linkedin.com/in/acmarpu

Source content: Google

https://github.com/acmarpu

Version Control System

- VCS is practice of tracking and managing changes done to a set of text files i.e Source code, scripts, configuration files.
- As we know that a software product is developed in collaboration by a group of developers they might be located at different locations and each one of them contributes to some specific kind of functionality/features. So, to contribute to the product, they made modifications to the source code (either by adding or removing). A version control system is a kind of software that helps the developer team to efficiently communicate and manage(track) all the changes that have been made to the source code along with the information like who made and what changes have been made.

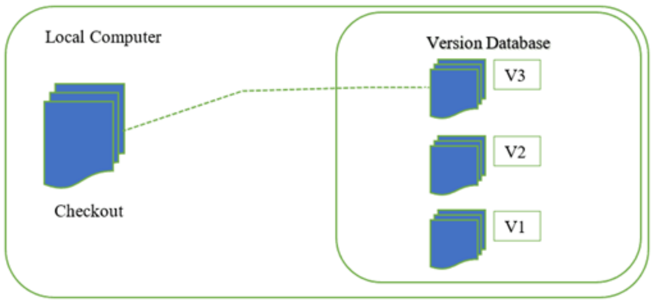
VCS Helps:

- A complete long-term change history of every file.
- Enhances the project development speed by providing efficient collaboration
- Helps in recovery in case of any disaster or contingent situation,
- Authorization

Types of VCS:

Local Version Control Systems:

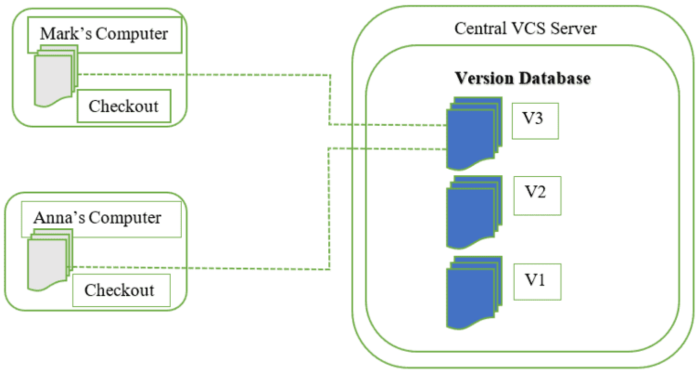
- A local version control system is a local database located on your local computer, in which every file change is stored as a patch. Every patch set contains only the changes made to the file since its last version.



- The main problem with this is that everything is stored locally. If anything were to happen to the local database, all the patches would be lost. If anything were to happen to a single version, all the changes made after that version would be lost.

Centralized Version Control Systems

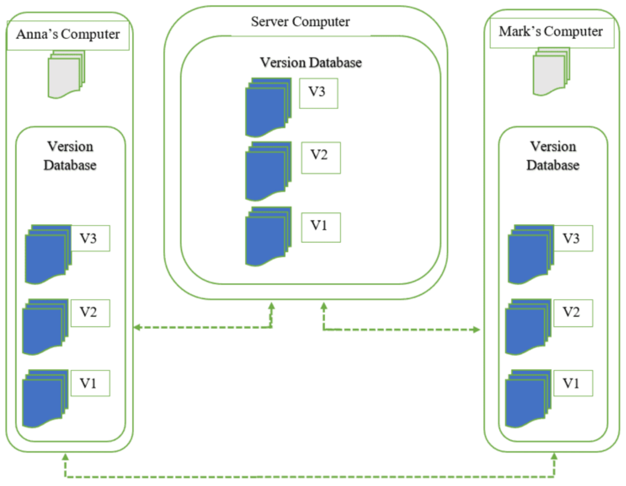
- A centralized version control system has a single server that contains all the file versions. This enables multiple clients to simultaneously access files on the server, pull them to their local computer or push them onto the server from their local computer.



- The biggest issue with this structure is that everything is stored on the centralized server. If something were to happen to that server, nobody can save their versioned changes, pull files or collaborate at all. Similar to Local Version Control, if the central database became corrupted, and backups haven't been kept, you lose the entire history of the project except whatever single snapshots people happen to have on their local machines.

Distributed Version Control Systems

- A Distributed Version Control System (DVCS) is a type of version control system where every developer's working copy of the codebase is a full repository that includes the complete history of all changes. Unlike centralized version control systems (like SVN), which rely on a single, centralized repository, DVCS repositories are distributed among multiple locations and can be synchronized with each other.



- Git is the most well-known example of distributed version control systems.

Git:

- Git is a tool used for version controlling system to track changes from small to large projects with speed and efficiency.
- It is very popular version control system that is used to manage projects.
- A version control system is software that is used to track changes made to our project and it stores these modifications as different versions of our project.
- Git is a version control system used to track changes in computer files. it is a tool to manage your code & file history while coordinating work remotely on those files with others.
- git is simple tool which can install in local system, which is acting as a local repository.
- A major advantage of using Git, or any other version control system, is that we can easily compare the current version of our project with previous versions and rectify errors. We can also roll back to the previous versions.
- It is a distributed version control system so that anyone who clones our repository will have access to the entire commit history.
- Other similar tools: Subversion, Mercuiral, AWS Codecommit, Azure DevOps Server.

How does it work?

- Git manages the different versions by maintaining a series of snapshots of the project. Whenever we want to commit or store a change permanently a new snapshot of our current version is taken and stored by Git.

GitHub:

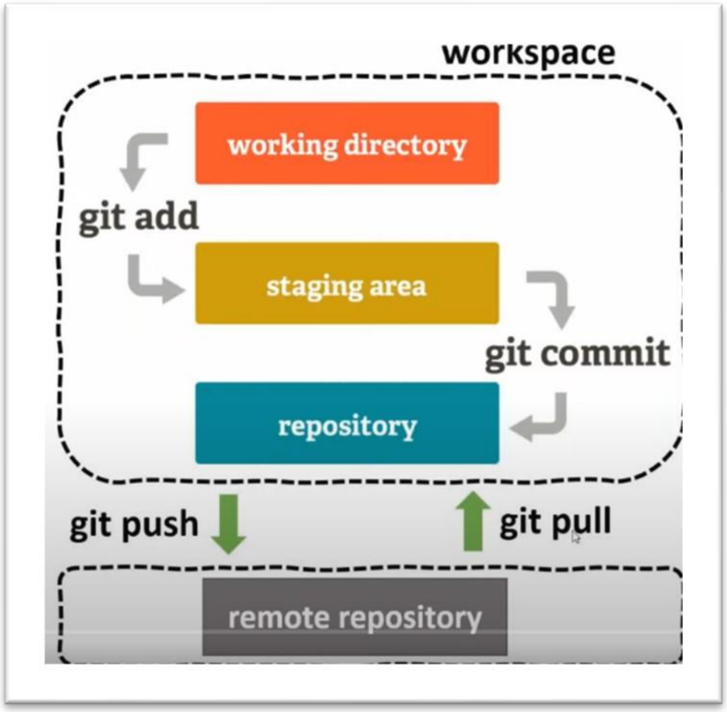
- GitHub is a hosting service for git repositories.
- Git is the tool, while GitHub is the service to use git
- GitHub is service which is acting as a remote repository
- GitHub is a Git Repository hosting service that is owned by the tech giant Microsoft. We can upload our local Git repository to GitHub and access it anywhere.

Disadvantages:

- Git provides mediocre support for binary files like images and multiple changes to such files make Git a bit cumbersome to use.
- Git has a steep learning curve and one needs to put in the time and effort to master it.
- Git is somewhat slower on the Windows platform as compared to other platforms.

Git Architecture/Workflow

- Working with Git consists of moving between three stages. Let us look at these three stages.
- **Working Directory** ---> **Staging** ---> **local repo** ---> **Remote Repo**



Working Directory:

- Refers to the actual files on the file system
- The place where we can make changes to our project like adding new files and deleting or modifying the existing files. Whenever we make any change to our files we move back to this stage.

<Untracked files>

Staging Directory

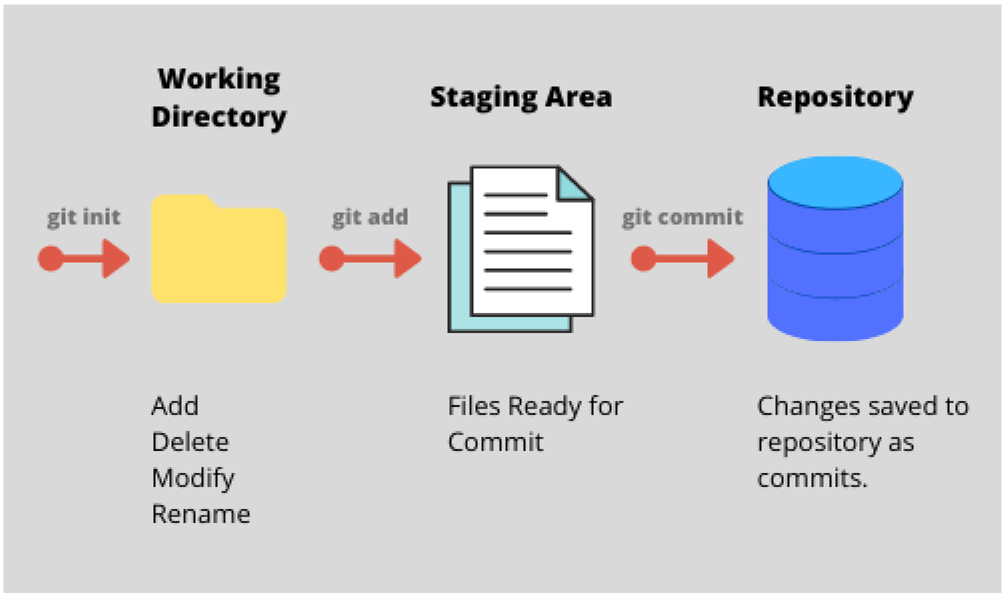
- Virtual area for creating snapshot
- Working directory or workspace we cannot directly send them to the local repository that means we cannot directly commit them into local repository so it should go through staging so staging is another layer which is basically a virtual layer between our workspace and your git local repository tracked files.
- Staging is the process of organizing and preparing our project files for a commit.

- Stage Changes: After making modifications, you add these changes to the staging area using

```
$ git add <file>          # Stages a single file
$ git add .               #Add all changes or all files
$ git add *.js            # Stages with a pattern
$ git ls-files --stage    # show staging area files
```

Git Local repository

- hidden dir which stores all the metadata.
- Once your files are there in the local repository, we can call them as a committed file or committed files
- The place where Git permanently stores or "commits" these changes as different versions of our project. All the snapshots of the project versions are stored here along with some additional information.



- Git works on a Distributed Model which means that all changes and updates to the repository will be visible to anyone who clones it. However, the owner has the control to make the repository public or private.
- Git provides excellent branching and merging features.

```
$ git commit -m <message>     # Commit with One line message
$ git commit                  # Opens the default editor to type a long message
$ git commit -am "Message"    # Skipping staging area
```

Git Config

- Git Config (short for configuration) is a command that helps us in setting various fields in Git like our name, email address, the default editor that should be used, etc. This is often done after installing Git on our system but these fields can also be changed at some other time in the future.
- The three levels of configuration are local, global, and system.

Local Git Config

- We can change the configuration at the local level i.e. the current Git Repository on which we are working.
- To change any fields at the local level we use the --local option along with the Git Config command. These settings reside in the **.git** directory by the name of config. We can view these configured values using the following command.

```
$ git config --list –local    # option specifies that we want to see the local repository-specific configuration

$ git config --local user.name "acmarpu"    # Adding User Name

$ git config --local  user.email "acmarpu@gmail.com"  # Adding email
```

Global Git Config

- We can set some global settings for our Git Repositories. These configurations will apply to all the git repositories for a particular user of a system.
- We use the --global option to alter these configurations. All these settings are stored in a .gitconfig file in the home directory

```
$ git config --global  user.email "acmarpu@gmail.com"  #configuration command

$ git config --global  user.name "acmarpu" #configuration command

$ git config --list –global # see the Global repository-specific configuration
```

System Git Config

- We can also configure the Git Repositories at the system level. These settings will apply to all the Git Repositories of all users that share a system. We use the --system option to change these values. These settings are stored in the <git-install-root>\mingw64\etc\gitconfig file.

```
$ git config --list –system
```

Branching

- Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes
- A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process

```
$ git branch <branch name>    # to create new branch
$ git branch                  # only show active branches
$ git branch –list            # List the branches
$ git branch -a               # show all branches
$ git checkout <branch name>    # switch to from one branch to another
$ git checkout -b <branch name>    # creating & switching to branch
$ git push --set-upstream origin <branch name>    # push created branch
$ git checkout -D branch name     #delete the branch
$ git branch -m <branch>    # Rename the current branch to <branch>.
$ git push origin main    # Git to push the changes from your local
$ git pull origin master # used in Git to fetch and integrate changes from a remote repository
$ git push --set-upstream origin <branch name> # is used in Git to push your local master branch to a remote repository
```

push your local repo into remote, create a local repo and remote repo copy url and add into local

```
$ git remote add origin <url> # is used to push the current branch to a remote repository
$ git remote add origin <https:>    # is used in Git to add a remote repository as a new remote named
$ git push –u origin –all    # is used in Git to push all local branches (including newly created ones) to a remote repository named
```

Creating remote branches

- git branch command also works on remote branches. To operate on remote branches, a remote repo must first be configured and added to the local repo config.

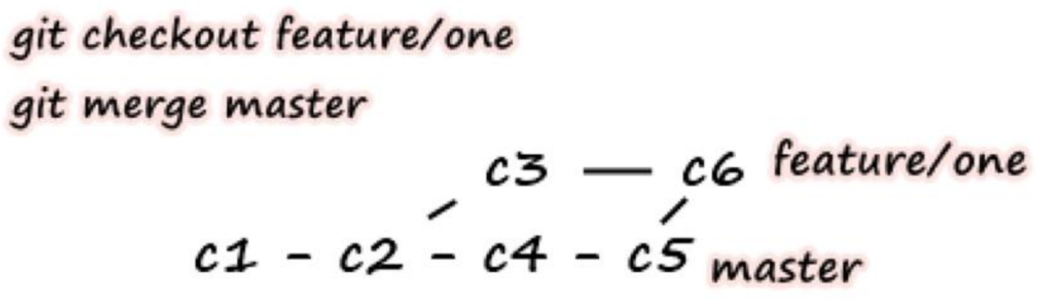
```
$ git remote add <new-remote-repo-name> https://<branchname> #Add remote repo to local repo config

$ git push <new-remote-repo> <name> # pushes the crazy-experiment branch to new-remote-repo

$ git remote -v
```

Git Merge

- Both git merge and rebase, kind of perform the similar task of combining/joining the development histories together.
- will Keep all commits history of the feature branch and move them into the master branch will add extra dummy commit

Perform Merge.

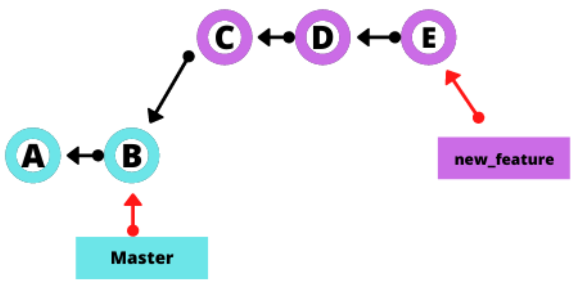
- Merging is the process of integrating multiple branches into one
- You are at your future branch and performed “get merge master” to merge the changes of master branch into your feature branch. So, to merge the changes a new merge commit ‘c6’ got created which binds histories from both the branches on feature branch.
- So, if you now do a “Git log” you will see a chronological order of commits
- **For e.g. ・ c1 – C2 – C3 – C4 – C5 – C6 (assuming C3 commit was done prior to C4 wrt time)**



```
git checkout feature/one
git merge master
```

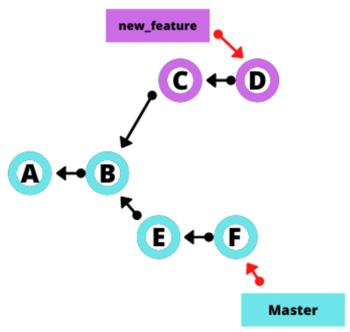
Fast-Forward Merge

- This type of merge strategy is used when the two branches are on the same linear path.
- For a fast forward merge to work the two branches must not diverge from a single point. The branch from which a new branch was created should not have added any new commits. In this type of merge Git simply takes the branch which is lagging in the linear path and sets it equal to the reference of the leading branch. It is also sometimes called the Two-Way Merge as it only involves two commits - the most recent ones from each branch.



Three-Way Merge

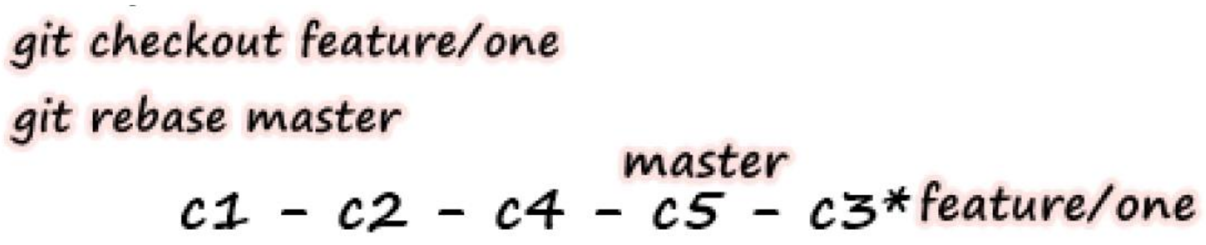
- Git uses this strategy when the two branches have diverged from a common ancestor. Both branches may have added some commits after splitting from that ancestor, so a linear path does not exist between them. Git performs this merge by first finding the common ancestor of both the branches and it then combines their individual commits into a single Merge Commit. This merge commit will have two parents, one for each branch. This helps in reverting back to any commit point of any one of the two branches. This method is called Three-Way Merge as it involves three commit points - the two most recent commits from the two branches and also an additional merge commit that is created.



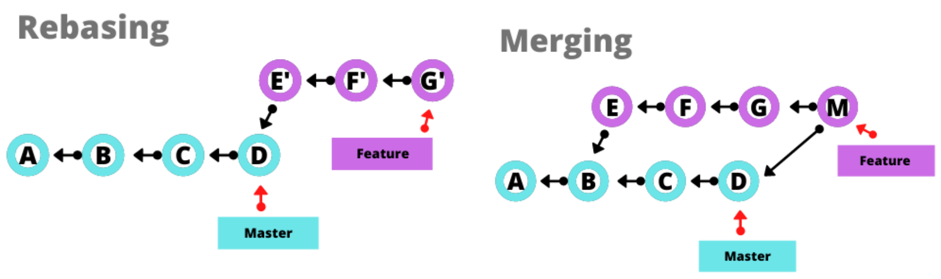
Git Rebase

- Will append all commits history of the feature branch in the front of the master branch Will NOT add extra dummy commit
- the git rebase command says "Reapply commits on top of another base tip," which literally means reapply commits on top of another base tip.
- The difference is that Git Merge will combine the changes of the two branches into a single merge commit while rebasing will just change the base of the branch to point to the latest commit of the other branch.
- Rebasing, on the other hand, maintains a linear commit history which makes
- it very easy to understand the project history.

Perform Rebase

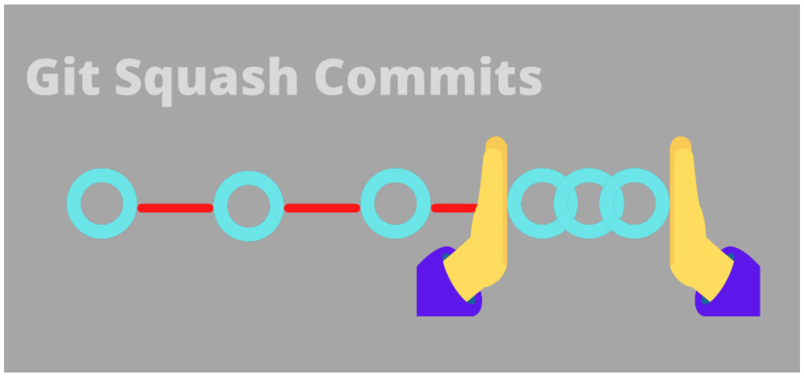


```
git checkout feature/one
git rebase master
```



Squash Merge:

- Will Group all feature branch commits into one commit then append it in the front of the master branch will add extra dummy commit.
- In Git terminology, squash means to merge or combine multiple commits into a single one. Git does not provide us with a dedicated command to squash commits. We can use the interactive Git Rebase command to squash commits.
- Squashing is usually done to keep our commit history clean and easy to understand.



Semi Linear Merge:

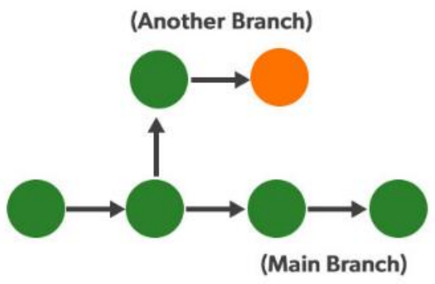
- it is a mix of rebase and a merge. first the commits in the pull request are rebased on top of the master branch.
- Then those rebased pull requests are merged into master branch
- in Git Flow, when you merge release into master, or master back into develop you need to retain any of the merge commits from PRs on those branches. The rebase portion of semi-linear merge will rewrite those merge commits, and not only will the "bubbles" be popped, but worse, the commit IDs will be rewritten

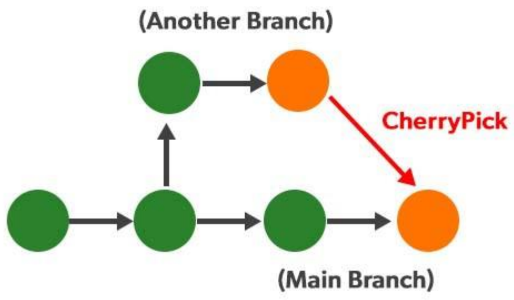
Chery pick

- Select individual commit to copy
- Cherry-picking is the process of selecting a commit and adding that commit to some other branch
- Unlike merging or rebasing that applies the changes of an entire branch to some other branch, Git Cherry-Pick just applies the changes of a single commit. It is mostly used when we have wrongly committed to some other branch or when a feature involved in one commit is also needed somewhere else

```
$ git cherry-pick <commitID>
```
This command allows you to select a commit from another branch

```
$ git cherry-pick --edit <commit-hash>
```
#Command in Git allows you to cherry-pick a commit and edit the commit message before finalizing the cherry-pick operation





Git Merge Tool

- Merge tool config
- is a Git command used to invoke a graphical or command-line tool for resolving merge conflicts during the merging of branches.
- Graphical or Command-line Interface: Depending on your Git configuration and the tools installed on your system, git merge tool can open a graphical merge tool (like meld, KDiff3, Beyond Compare, etc.) or a command-line tool (like vimdiff, emacs, etc.) to help you resolve conflicts.

$ git config --global merge.tool <tool-name> # Replace <tool-name> with the name of the merge tool you want to use

$ git config –global merge.tool p4merge # Using p4merge tool

$ git config –global mergetool.p4merge.path # you need to set up the path to the P4Merge executable in your Git configuration

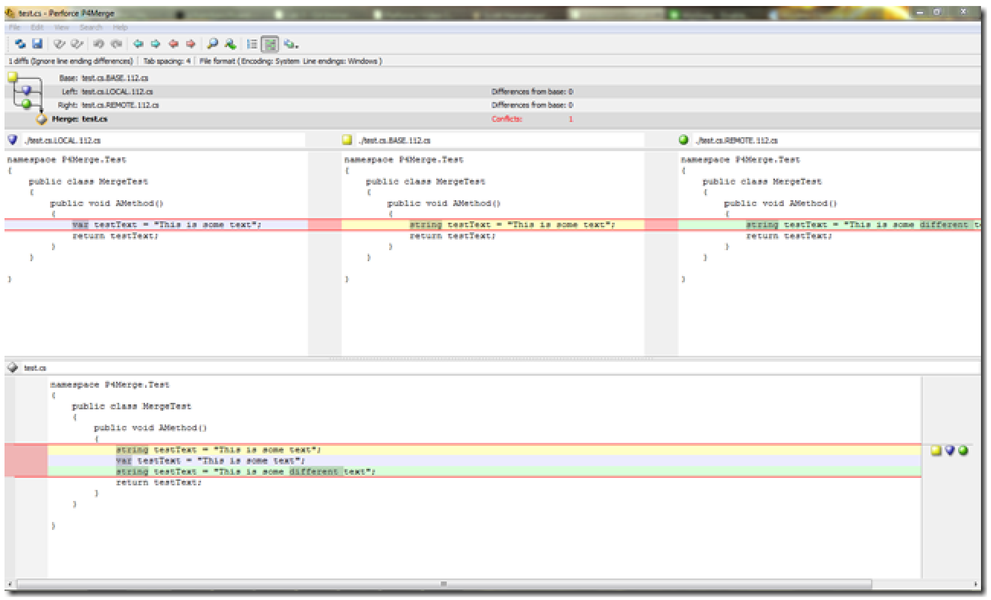
$ git config –global mergetool.prompt false # mergetool.prompt to false disables the prompt that asks for confirmation before running a merge tool.

Diff tool config

$ git config –global diff.tool p4merge # is used in Git to configure P4Merge as the default tool for viewing file differences (diffs) between different versions of files in your Git repositories

$ git config – global difftool.p4merge.path # path to p4merge"

$ git config –global difftool.prompt false


test.cs - Perforce P4Merge
File Edit View Search Help
1 diffs (Ignore line ending differences) | Tab spacing: 4 | File format (Encoding: System Line endings: Windows)
Base: test.cs.BASE.112.cs
Left: test.cs.LOCAL.112.cs
Right: test.cs.REMOTE.112.cs
Merge: test.cs
Differences from base: 0
Differences from base: 0
Conflicts: 1
./test.cs.LOCAL.112.cs
./test.cs.BASE.112.cs
./test.cs.REMOTE.112.cs
test.cs

```
namespace P4Merge.Test
{
    public class MergeTest
    {
        public void AMethod()
        {
            var testText = "This is some text";
            return testText;
        }
    }

}
```

```
namespace P4Merge.Test
{
    public class MergeTest
    {
        public void AMethod()
        {
            string testText = "This is some text";
            return testText;
        }
    }

}
```

```
namespace P4Merge.Test
{
    public class MergeTest
    {
        public void AMethod()
        {
            string testText = "This is some different t
            return testText;
        }
    }

}
```

```
namespace P4Merge.Test
{
    public class MergeTest
    {
        public void AMethod()
        {
            string testText = "This is some text";
            var testText = "This is some text";
            string testText = "This is some different text";
            return testText;
        }
    }

}
```

Git diff

- Diff is short for the word difference
- Git Diff is a command that helps us to see the changes that we have made to a particular entity. We can compare the changes made to files, commits, or even different branches. When working with multiple files, we may forget the changes that we have made since the last commit or we just want to check the changes that we have made before finally committing them. Git Diff helps us in doing all this.
- to compare the file contents between working directory or staging or between working directory or local repository or between local repository remote repository we can use git get diff command basically used for comparing the files.
- Ex1: To view the changes made to a file since the last time it was staged

  ```
  $ git diff index.txt
  ```

  *Output if any line prefixed with space means it is unchanged
  *Output if any line prefixed with +means it is added in destination copy
  *Output if any line prefixed with -means it is removed from destination copy

- Ex2: To see the difference in file Content between Working Directory and last commit

  ```
  $ git diff HEAD index.txt #Last commit referred using HEAD
  ```

- Ex3: To see the difference in file content between staged copy and last commit
  compares the changes between the version of index.txt in the staging area

  ```
  $ git diff --staged HEAD index.txt
  ```

- Ex4: To see the difference in file content between specific commit working directory copy

  ```
  $ git log --oneline
  $ git diff d0ff720 index.txt
  ```

- Ex5: to see the difference in file content between specific commit and staging are copy

  ```
  $ git diff --staged d0ff720
  ```

- Ex6: To see the diffrence in file content between 2 specified commits

  ```
  $ git diff d0ff720 c0ff720 index.txt
  ```

- Ex7: To see the differences in content between 2 branches
  Master and other branches

  ```
  $ git diff <master> <test>  # it shows all differences between master branch and test branch
  ```

- Ex8: to see the differences in content between Local and remote Repositors

  ```
  $ git diff master origin/master
  ```

- Comparing Files Between Two Branches Using Git Diff

```
$ git diff <branch-1>..<branch-2> filename
```

- Comparing Commits Between Two Branches Using Git Log

```
$ git log <branch-1>..<branch-2>
```

Git Init

- Git init is going to be the very first command that we will always use when working with Git.
- It is used to initialize a new Git Repository and tell Git to track any changes that we make in that repository. We only need to do this once while creating a new Git Repository. To convert an existing directory into a Git repository we need to move to that directory and just run the git init command directly.
- Let's take a look at how to use the git init command.

```
$ git init
```

Git Clone

- To clone a remote repository to our local system, we need the URL to that remote repository. Let's take a look at how to use git clone command.

```
$ Git clone "github url"
$ git clone <remote-repository-url> <directory-name>
```

- The example shown below will clone only the last 4 commits that were made to the project.

```
$ git clone -depth=4 <remote-repository-url>
```

Git Add

- git add command that allows Git to track a particular file and check for any modifications to it.

```
$ git add 1.txt    #adding file from workspace to staging area
$ git add .*  #adding all . files which are started with. from workspace to staging area
```

Git status

- it will show files status which are available in workspace

```
$ git status
```

Git Commit

- Git does this by capturing a snapshot of the state of the files present in the Staging Area and storing it permanently in the repository as a new version of the project.
- At any point in time, we can move back to any of these versions that we committed.

```
$ git commit -m "this is my first commit" #commit to local repo
$ git commit -a -m "both the files are modified"
```

Viewing a Commit

$ git show 921a2ff #show the given commit

$ git show HEAD #Shows the last commit

$ git show HEAD~2 #Two steps before the last commit

$ git show HEAD:file.js # Shows the version of files.js stored in the last commit

Git .gitignore file

- The .gitignore file is a text document that tells Git what files it should ignore. We don't always want Git to keep a track of every file on our repository and we can ask Git to ignore the changes that we make to our files with the help of a special .gitignore file.
- By ignoring we mean that Git should not track any changes made to these files.

Git Log

- Git is used to save different versions of our project in the form of snapshots stored in commits. The Git log command is used to view the commit history of our repository.

$ git log #it will show how many commits you have done earlier

$ git log -1 #it shows the latest commit

$ git log --oneline #it will show in one shot

$ git log --stat #It will tell us how many lines were added and deleted to the files.

$ git log --graph #command to view the commit history in the form of a graph

$ git log --after="YYYY-MM-DD" #To view commits made after a certain date

$ git log --before="YYYY-MM-DD" #To view commits made before a certain date

$ git log --after="YYYY-MM-DD" --before="YYYY-MM-DD" #To view commits made between two dates

$ git log --author="author-name" #commits made by a certain author by using the --author option.

$ git log --grep="commit-message" #commits that have the matching words in their commit message

$ git log – -oneline --all –graph # it shows graph view

git rm

- git rm remove files from both staging & working directory

```
$ git rm file.txt

$ git rm <file-1> <file-2> <file-3>

$ git rm -r .
```

`$ git rm --cached <file-name>` # We can also remove files from just the staging area by using the **--cached** option.

`git rm --dry-run <glob-pattern>` # There is an option to dry-run a Git Rm command. This won't remove any files and can be useful in situations where you just want to see the files that will be removed by the Git Rm command

How to Reverse a Removal?

- There may be cases when we remove a file and then again need the same file back. We can undo the effect of the Git Rm command as long as we have not committed the changes by removing the file.
- Git reset command can be used to go back to the last commit (HEAD).

```
$ git reset head
```

`$ git reset 1.txt` #undo operations from git staging area

- Move the HEAD to the specified commit, and all remaining recent commits will be removed
- mode will decide whether these changes are going to remove from staging area and working director or not

```
$ git reset <mode> <commit id>
```

--mixed

--soft

--hard

1) git reset with --mixed mode

- --mixed (default): This mode moves the HEAD to the specified commit and resets the index (staging area) to match the commit, but it does not affect the files in the working directory. This means all changes are unstaged after this reset.

```
$ git reset --mixed <commit id>
```

2) git reset with --soft mode

- --soft: This mode moves the HEAD to the specified commit while keeping all changes in the index (staging area) and working directory as staged changes. It does not change any files in the working directory.

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

3) git reset with --hard mode

- --hard: This mode moves the HEAD to the specified commit, resets the index to match the commit, and resets the files in the working directory to match the commit. It discards all changes made after the specified commit.
- it is impossible to revert and hence while using hard reset we must take special care

```
$ git reset --hard <commit id>
```

4) Checkout command

- We can use checkout command to discard unstaged changes in the tracked files of working directory

```
$ git checkout -- 1.txt   #undo operations
```

1) only for working directory
2)to discard unstaged changes   #The changes which are not added to staging area)
3) in the tracked files   #The files which are already added to staging area/commit)

Git Remote

- A remote repository (also called remote) is a shared central repository where developers can push and pull changes. Git provides us with the Git Remote command which can be used to work with remote repositories.
- To remove a remote, use the rm option with the Git Remote command and pass the name of the remote that you wish to delete.

```
$ git remote rm <remote-name>
```

```
$ git remote rename <old-remote-name> <new-remote-name>
```
#We can also rename an existing remote by using the rename

- Use the set-url option to change the URL of an existing remote

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

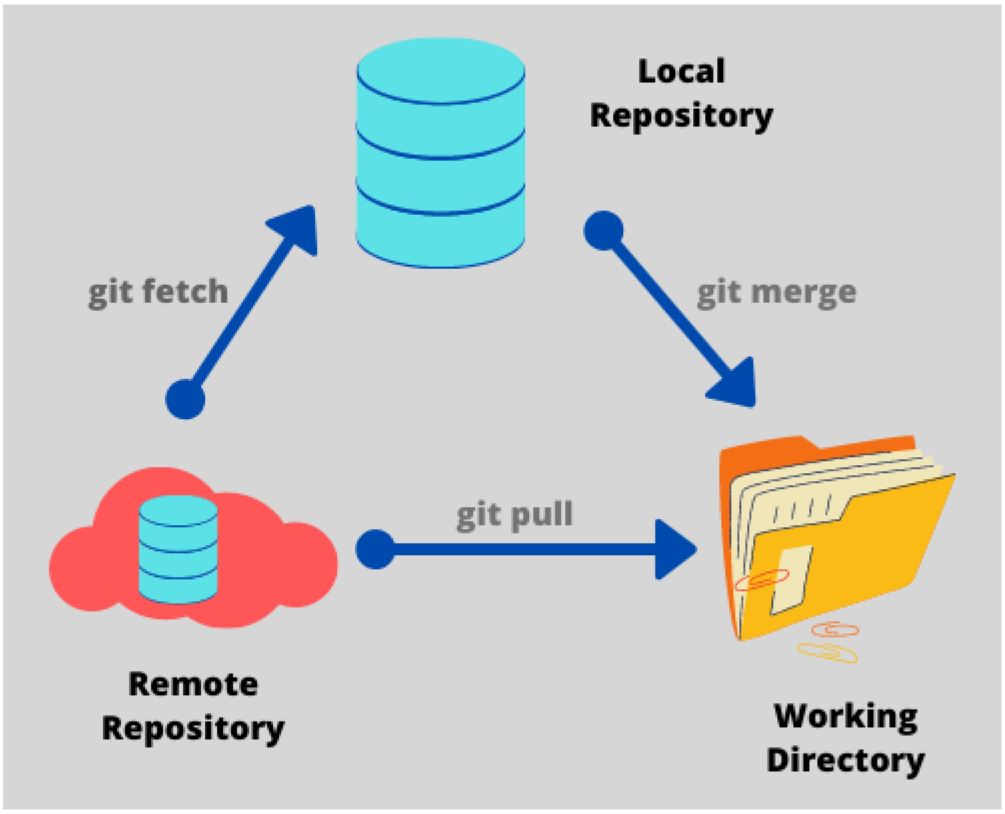
Git Fetch

- Git Fetch is a command that lets us download the changes from a remote repository and to our local repository but stores them as a separate branch which is referenced by using the remote-tracking branches. It keeps the local master branch unchanged.
- Git Fetch is a very commonly used command to communicate with the remote repository. Git Fetch only downloads the new commits in a separate branch without merging them with the branch on which we are currently working.

```
$ git fetch <remote-repository-url> # To fetch the entire remote repository
```

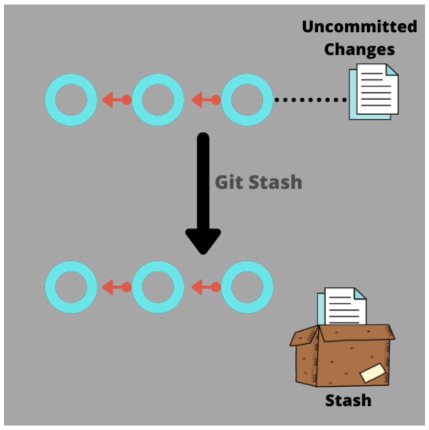
```
$ git fetch <remote-repo-url> <branch> #To fetch just a single branch from the remote repository:
```

```
$ git fetch –all #To fetch all the branches from the remote repository
```



Stashing

- Stashing is the process of storing unsaved changes of our project in a separate location. After stashing, all the changes that were not yet committed are temporarily stored.



```
$ git stash
```
#To simply stash the changes

```
$ git stash save "Message"
```
#We can add a message to our stashes by using the save option

```
$ git stash list
```
#Viewing a Stash

Retrieving the Stashes

- Just stashing away the changes is not enough. We should be able to get those stashed changes back. Git provides us a few different options to do that.
- To get the last stashed change back we can use the apply option.

  ```
  $ git stash apply
  ```

  ```
  $ git stash apply stash@{stash-index}
  ```
  #We can also get the changes saved to any previous stash by using its stash id.

- We can also view all the changes made to the files using the **-p** flag along with the Git Stash Show.

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

- We can also apply the stash changes and delete the saved stash by using the pop option. If no stash id is provided then it will pop and apply the most recent stash.

  ```
  $ git stash pop stash@{stash-index}
  ```

- We can also directly remove a stash without applying it by using the drop option with Git Stash:

  ```
  git stash drop stash@{stash-index}
  ```

- We can delete all the stashes with a single command by using the clear option.

  ```
  $ git stash clear
  ```

Tagging

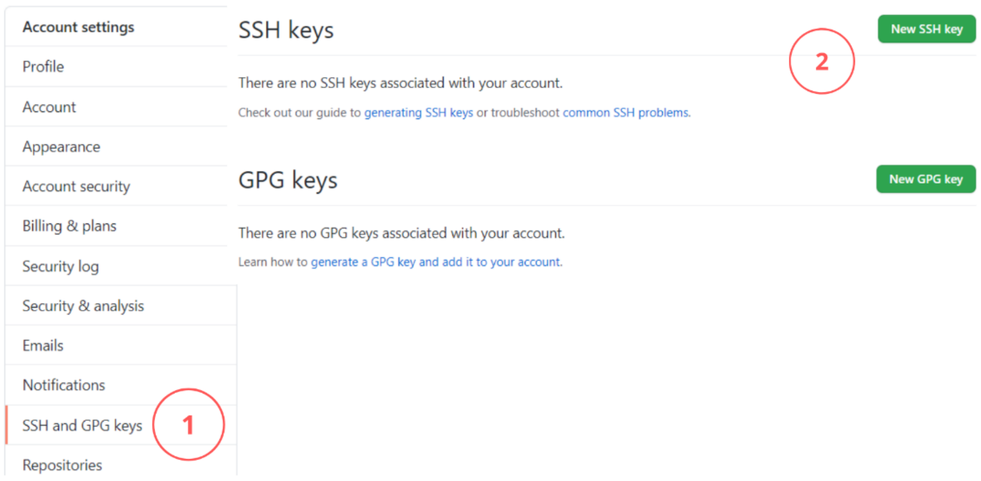
- Tagging is a way of marking important points in the history of our project so that we can reference them in the future. In Git tagging is done by using the Git Tag command.
- Tags are mostly used to mark software release versions or to mark a specific commit that we may need in the future.
- There are two types of tags, Annotated and Lightweight. To create an annotated tag we need to use the -a flag but we can create a lightweight tag simply by giving the name of the tag to the command. This will create a new tag at the point where the HEAD is pointing.

```
$ git tag <tag-name>
$ git tag -a <tag-name>
$ git tag -a <tag-name> -m "Message"
$ git show <tag-name>
$ git tag -d <tag-name> #Delete the tag name
```

How to Generate Git SSH Keys

- SSH Keys are a secure and easy way of connecting to the remote repository. They help the remote repository identify the user who is making a push and are convenient for the users as they don't have to manually enter their username and password every time. Let's learn more about SSH keys and how to use them.
- Generating SSH Keys in Windows using Putty Gen
- In Linux

```
$ ls -al ~/.ssh
$ mkdir $HOME/.ssh
$ ssh-keygen -t rsa -b 4096 -C "email"
$ eval "$(ssh-agent -s)"
$ ssh-add ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub
```



SSH keys

There are no SSH keys associated with your account.

Check out our guide to generating SSH keys or troubleshoot common SSH problems.

GPG keys

There are no GPG keys associated with your account.

Learn how to generate a GPG key and add it to your account.