# Version Control

Basically saving the project at different time points (save points), each time the code is updated it is a new version, and basically you have upgraded.

This helps to keep save points and if at some point the code doesn’t work as expected, then you can go back to previous save point.

You can also compare the code line by line, where you messed.

# Git working in Action

So, Github is basically controlling and having various save points and checkpoints so that you can go back any day and retrieve the last saved changes. So, Github can be said as a four step process.

1. Working directory (or the working tree)
   1. This is where the work is being done at the most root level and being saved to.
   2. At this stage, nothing is committed to repositories and basically everything is being done without having any save points and checkpoints.
   3. The working area is where files that are not handled by git. These files are also referred to as "untracked files."
   4. Here if you make changes and do not explicitly save them to git, you will lose the changes made to your files. This loss of changes occurs because git is not aware of the files or changes in the Working Tree until you tell it to pay attention to them. If you make changes to files in your working tree git will recognize that they are modified, but until you tell git “Hey pay attention to these files,” it won’t save anything that goes on in them.
2. Staging area
   1. Like in the theatre, we have the staging area behind the main stage, in git we have this staging area.
   2. Staging area is files that are going to be a part of the next commit, which lets git know what changes in the file are going to occur for the next commit.
   3. It is at the stage when we have added the changes to git after using git add command and before the git commit command.
   4. The Staging Area is when git starts tracking and saving changes that occur in files. These saved changes reflect in the .git directory. That is about it when it comes to the Staging Area. You tell git that I want to track these specific files, then git says okay and moves them from you Working Tree to the Staging Area and says “Cool, I know about this file in its entirety.” However, if you make any more additional changes after adding a file to the Staging Area, git will not know about those specific changes until you tell it to see them. You explicitly have to tell git to notice the edits in your files.
3. Local repository
   1. The Local Repository is everything in your .git directory. Mainly what you will see in your Local Repository are all of your checkpoints or commits. It is the area that saves everything (so don’t delete it). That’s it.
   2. Now, the characters of the play have been brought to stage so that the audience can remember it forever. The play is being recorded live via a camera.
4. Remote repository
   1. Once you made the changes, saved them locally, told git about the changes and then committed them locally, you take it to the cloud storage that is Github by committing it globally.
   2. We made a video of the play and uploaded on the internet so that it can be viewed globally and be saved forever.
   3. This is done for collaboration or making your code visible globally for various purposes such as education, or showing your resume.

# git config

Usage: git config –global user.name “[name]”

Usage: git config –global user.email “[email address]”

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

Git Config Command - Git Commands - Edureka

**git init**

Usage: git init [repository name]

This command is used to start a new repository.

GitInit Command - Git Commands - Edureka

**git clone**

Usage: git clone [url]

This command is used to obtain a repository from an existing URL.



**git add**

Usage: git add [file]

This command adds a file to the staging area.

Git Add Command - Git Commands - Edureka

Usage: git add \*

This command adds one or more to the staging area.

Git Add Command - Git Commands - Edureka

**git commit**

Usage: git commit -m “[ Type in the commit message]”

This command records or snapshots the file permanently in the version history.



Usage: git commit -a

This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then.

Git Commit Command - Git Commands - Edureka

**git diff**

Usage: git diff

This command shows the file differences which are not yet staged.



 Usage: git diff –staged

This command shows the differences between the files in the staging area and the latest version present.



Usage: git diff [first branch] [second branch]

This command shows the differences between the two branches mentioned.



**git reset**

Usage: git reset [file]

This command unstages the file, but it preserves the file contents.



Usage: git reset [commit]

This command undoes all the commits after the specified commit and preserves the changes locally.

Git Reset Command - Git Commands - Edureka

Usage: git reset –hard [commit]  This command discards all history and goes back to the specified commit.

Git Reset Command - Git Commands - Edureka

**git status**

Usage: git status

This command lists all the files that have to be committed.



**git rm**

Usage: git rm [file]

This command deletes the file from your working directory and stages the deletion.

Git Rm Command - Git Commands - Edureka

**git log**

Usage: git log

This command is used to list the version history for the current branch.



Usage: git log –follow[file]

This command lists version history for a file, including the renaming of files also.



**git show**

Usage: git show [commit]

This command shows the metadata and content changes of the specified commit.



**git tag**

Usage: git tag [commitID]

This command is used to give tags to the specified commit.



**git branch**

Usage: git branch

This command lists all the local branches in the current repository.

Git Branch Command - Git Commands - Edureka

Usage: git branch [branch name]

This command creates a new branch.

Git Branch Command - Git Commands - Edureka

Usage: git branch -d [branch name]

This command deletes the feature branch.

Git Branch Command - Git Commands - Edureka

**git checkout**

Usage: git checkout [branch name]

This command is used to switch from one branch to another.

Git Checkout Command - Git Commands - Edureka

Usage: git checkout -b [branch name]

This command creates a new branch and also switches to it.

Git Checkout Command - Git Commands - Edureka

**git merge**

Usage: git merge [branch name]

This command merges the specified branch’s history into the current branch.

Git Merge Command - Git Commands - Edureka

**git remote**

Usage: git remote add [variable name] [Remote Server Link]

This command is used to connect your local repository to the remote server.

Git Remote Command - Git Commands - Edureka

**git push**

Usage: git push [variable name] master

This command sends the committed changes of master branch to your remote repository.



Usage: git push [variable name] [branch]

This command sends the branch commits to your remote repository.



Usage: git push –all [variable name]

This command pushes all branches to your remote repository.



Usage: git push [variable name] :[branch name]

This command deletes a branch on your remote repository.



**git pull**

Usage: git pull [Repository Link]

This command fetches and merges changes on the remote server to your working directory.



**git stash**

Usage: git stash save

This command temporarily stores all the modified tracked files.

Git Stash Command - Git Commands - Edureka

Usage: git stash pop

This command restores the most recently stashed files.



Usage: git stash list

This command lists all stashed changesets.

Git Stash Command - Git Commands - Edureka

Usage: git stash drop

This command discards the most recently stashed changeset.

Git Stash Command - Git Commands - Edureka

# Branching and Merging

We may need to work on a project but not directly but say on an experimental setup. Since making copies to try out different things will take lot of space and time, we can make several branches for experimentation and if changes are good or work out, then we can merge the changes with the master branch, and if not then we can just delete or discard the changes like nothing ever existed.

We first make branches for this via git branch [branch\_name]

Then make changes, add them and commit them to that specific branch and if it seems good, then we can use command:

git merge [branch\_name]

from the branch where we want to receive changes from and enter the branch\_name for the branch we want changes coming from.

# Forking and pulling

Suppose, we have some repository that needs to be collaborated upon so that some changes can be made and new features can be implemented that is not us but some other person. This person’s changes might be good and all might just work out but if that is not the case then our code will be all messy if this person was to directly commit to the same repository.

In that case, the person should have his own copy, make changes on his own account or system and then suggest them to the original owner of the repository. This entire procedure is called forking and pulling.

The person forking first makes the copy of repository on his own Github account and then clone it for working locally. Then he makes changes locally, pushes them to the repository on his **own** Github account and then makes a pull request on the original repository. If the original owner of the repository likes it, then he/she can pull those changes for his own repository.