

# GIT - Global Tracking System

## Introduction to Git

In professional development environments, the code is written in teams of developers. From your colleague sitting around the corner to a remote developer living across the globe, anyone could be writing code with you and contributing to the same codebase. The best and most popular way to contribute code to a single codebase is Version Control Systems.

## What's Git?

Git is a distributed version control system (DVCS). "Distributed" means that all developers within a team have a complete version of the project. A version control system is simply software that lets you effectively manage application versions. Thanks to Git, you'll be able to do the following:

- Keep track of all files in a project
- Record any changes to project files
- Restore previous versions of files
- Compare and analyze code
- Merge code from different computers and different team members.

## Version Control Systems:

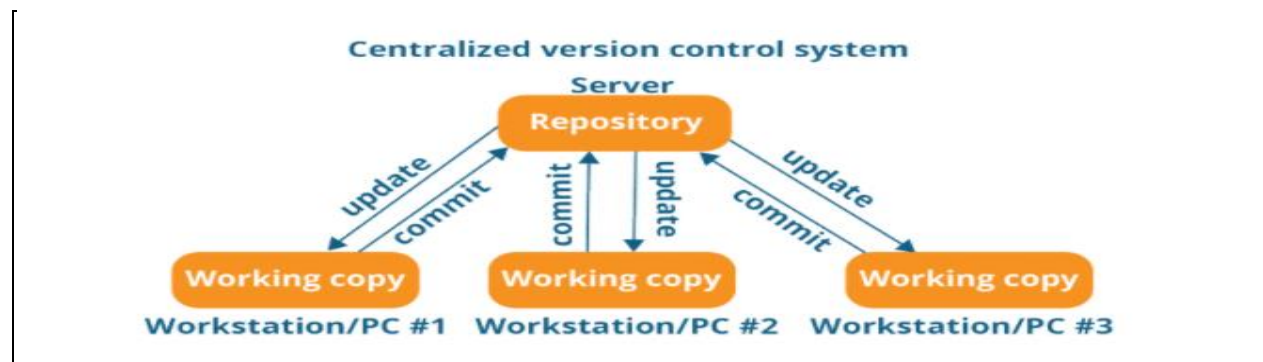
Version control systems are systems that allow us to contribute to and manage different releases and stages of a software product without actually having to keep multiple files or folders. They also make development within a team more manageable and less of a pain as developers don't have to be trading folders, but rather communicate with a single source where all the changes are happening and everything is saved.

There are two types of Version Control Systems: Centralized and Distributed.

### 1. Centralized Version Control Systems

Centralized means that the codebase or the project resides in one central place also known as a repository. In order to make changes, people need to have access to that repository and write their code in it.

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer's workstation.

Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

- It is not locally available; meaning you always need to be connected to a network to perform any action.
- Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

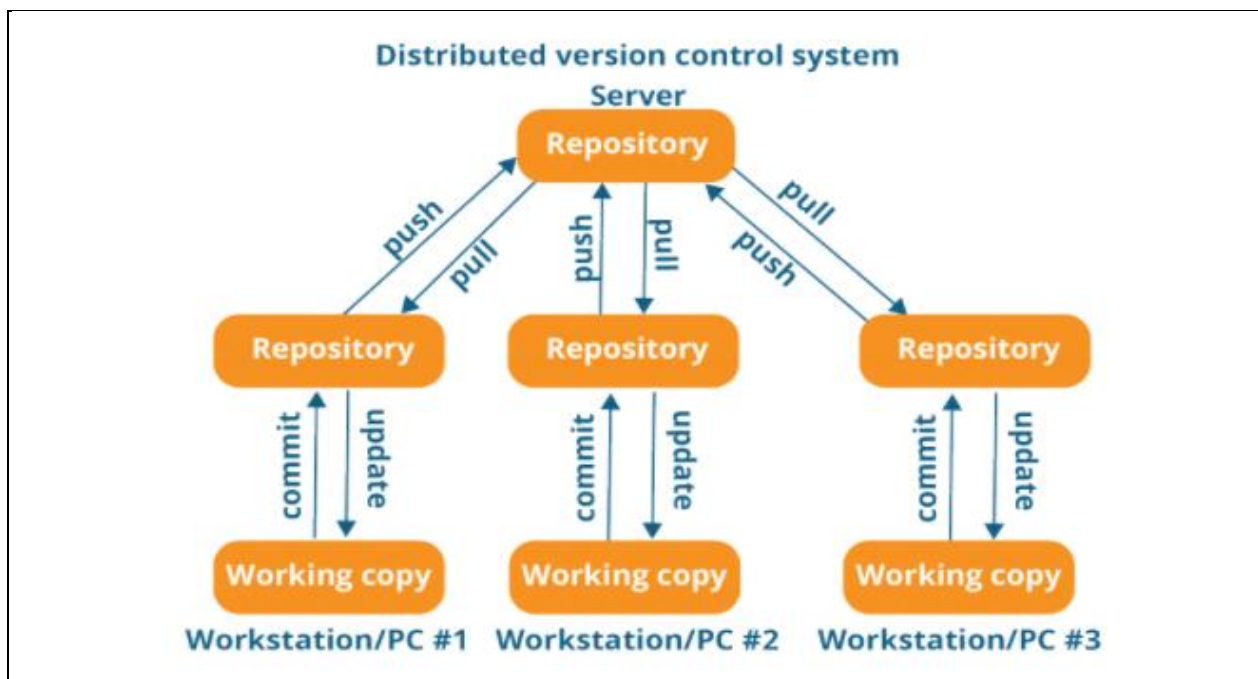
This is when Distributed VCS comes to the rescue.

## 2. Distributed Version Control Systems.

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

- All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
- Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.
- Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
- If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor’s local repositories.

## How Git works

Most version control systems work by calculating the differences in each file, and then by summing up these differences they can reach whichever version saved, but that’s not the way Git works.

Git functions as a series of snapshots for your file system. Every time you change something in your files committing it, or just saving the state of your project, Git takes a picture of the whole system and saves a reference to it. If there’s a file that hasn’t been changed, then it stores a reference to the previous snapshot. We’ll get to how this is one of the most powerful and enabling features in Git when we dive deeper into the features Git provides.

## Git Architecture and Stages:

There are four states for your files to be in when you’re working with Git. These three states are: **Working, staging, local and remote**

### Working Directory:

When you make changes to a file in your working directory, it is then modified, but these modifications are not stored in the codebase.

### Staging Area:

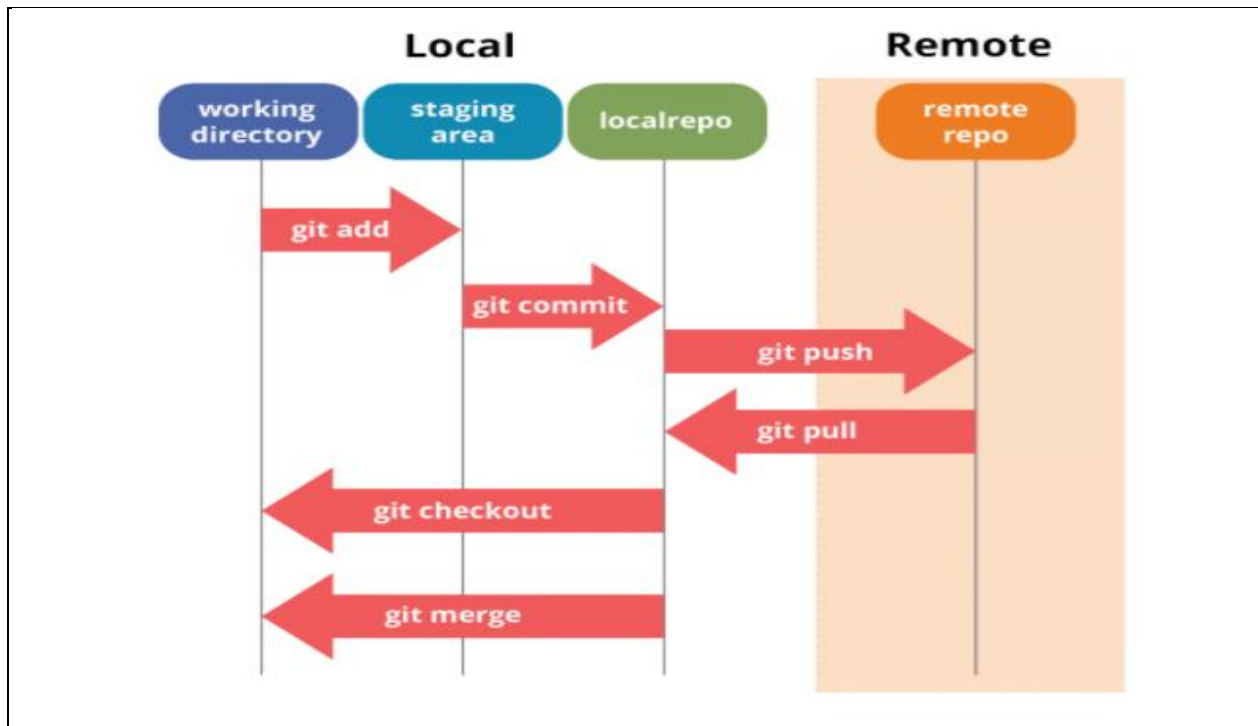
When you **stage** the file, these changes are marked to be saved in the next commit.

### Local Repo:

When you **commit** the file, these changes are finally stored in your codebase. This you have a local repository on your machine, that is your desktop or laptop, which is defined in your working directory

### Remote Repo:

A remote repository that is stored on the cloud. Eg: Github



## Git Installation

### Install Git Using Yum

1. Update yum repository

```
# yum update
```

2. Install git tool using yum command.

```
# yum install git
```

```

[root@BIVM ~]# yum install git
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
base: mirror.vanehost.com
extras: mirror.vanehost.com
updates: mirror.vanehost.com
Resolving Dependencies
--> Running transaction check
--> Package git.x86_64 0:1.8.3.1-21.el7_7 will be installed
--> Processing Dependency: perl-Git = 1.8.3.1-21.el7_7 for package: git-1.8.3.1-21.el7_7.x86_64
--> Processing Dependency: perl(Git) for package: git-1.8.3.1-21.el7_7.x86_64
--> Running transaction check
--> Package perl-Git.noarch 0:1.8.3.1-21.el7_7 will be installed
--> Finished Dependency Resolution
Dependencies Resolved

=====
Package Arch Version Repository Size
Installing:
git x86_64 1.8.3.1-21.el7_7 updates 4.4 M
Installing for dependencies:
perl-Git noarch 1.8.3.1-21.el7_7 updates 55 k
Transaction Summary
Install 1 Package (+1 Dependent package)
Total download size: 4.4 M
Installed size: 22 M
Is this ok [y/d/N]: y
Downloading packages:
(1/2): perl-Git-1.8.3.1-21.el7_7.noarch.rpm | 55 kB 00:00:02
(2/2): git-1.8.3.1-21.el7_7.x86_64.rpm | 4.4 MB 00:00:18
Total 241 kB/s | 4.4 MB 00:00:18
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Installing : perl-Git-1.8.3.1-21.el7_7.noarch 1/2
Installing : git-1.8.3.1-21.el7_7.x86_64 2/2
Verifying : git-1.8.3.1-21.el7_7.x86_64 1/2
Verifying : perl-Git-1.8.3.1-21.el7_7.noarch 2/2
Installed:
git.x86_64 0:1.8.3.1-21.el7_7

Dependency Installed:
perl-Git.noarch 0:1.8.3.1-21.el7_7
Complete!
[root@BIVM ~]#

```

### 3. Check the git version

```
# git --version
```

```

[root@BIVM ~]# git --version
git version 1.8.3.1
[root@BIVM ~]#

```

## Install Git from Source

---

Before you begin, first you need to install required software dependencies from the default repositories, along with the utilities that needed to build a binary from source:

```
# yum groupinstall "Development Tools"
# yum install gettext-devel openssl-devel perl-CPAN perl-devel zlib-devel
```

After you have installed required software dependencies, go to the official [Git release page](#) and grab the latest version and compile it from source using following series of command:

```
# wget https://github.com/git/git/archive/v2.10.1.tar.gz -O git.tar.gz
# tar -zxvf git.tar.gz
# cd git-2.10.1/
# make configure
# ./configure --prefix=/usr/local
# make install
# git --version
```

## Commands

- **git config**
- **git init**
- **git clone**
- **git add**
- **git commit**
- **git status**
- **git diff**
- **git log**
- **git revert**
- **git reset**
- **git rm**
- **git show**
- **git tag**
- **git branch**
- **git checkout**
- **git merge**
- **git remote**
- **git push**
- **git fetch**
- **git pull**
- **git stash**
- **git rebase**
- **git cherry-pick**

## 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 --global user.name "King Kong"
$ git config --global user.email "king-kong@gmail.com"
```

## git init:

**git init** creates an empty Git repository or re-initializes an existing one. It basically creates a **.git** directory with sub directories and template files. Running a **git init** in an existing repository will not overwrite things that are already there. It rather picks up the newly added templates.

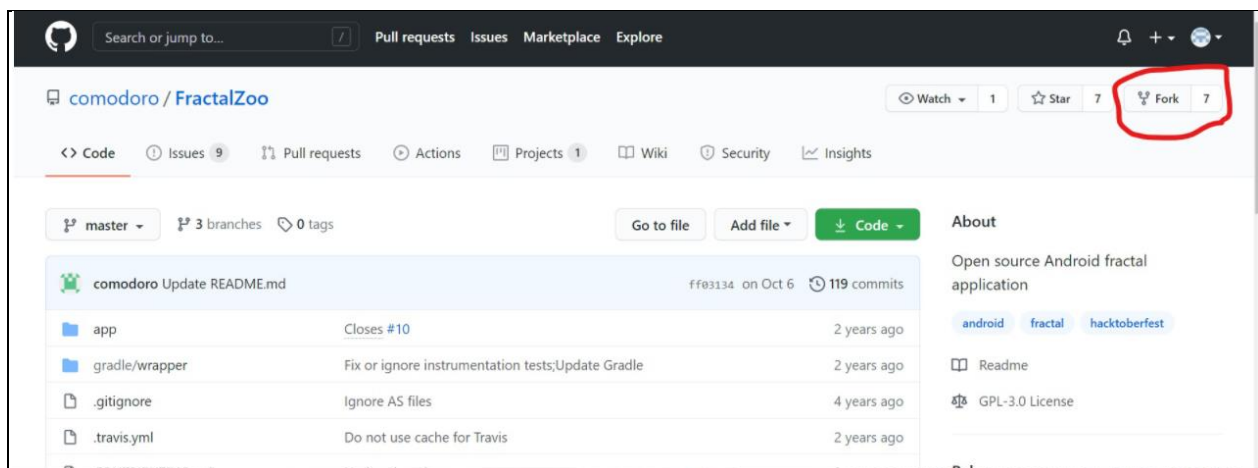
Usage: `git init [repository name]`

```
$ git config --list
user.name=King Kong
user.email=king-kong@gmail.com
```

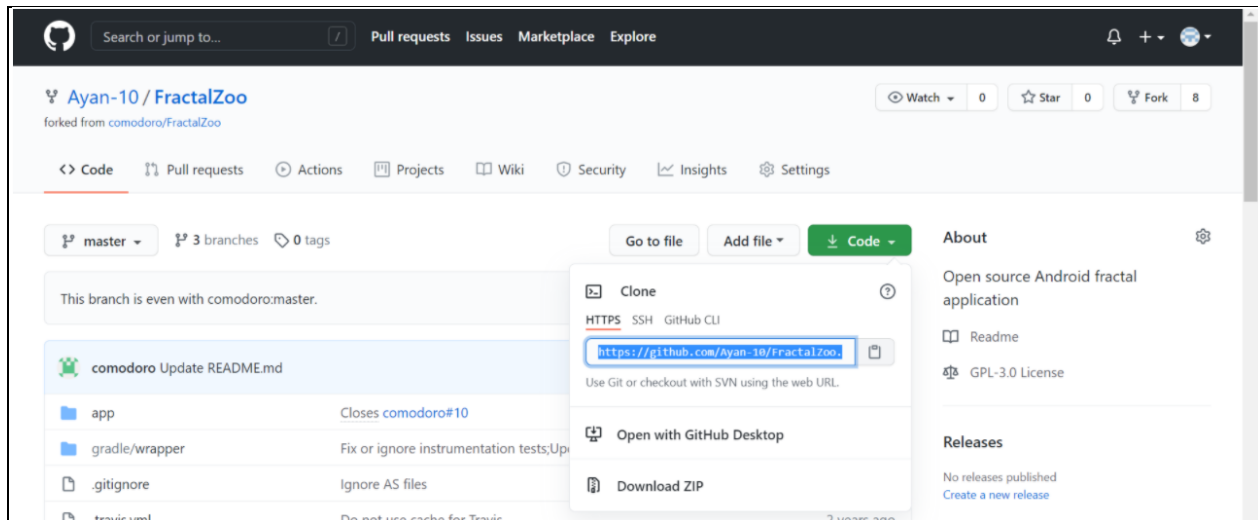
## git clone:

If you want to open-source contribution. First, you have to copy an existing repository (the repository, where you want to contribute) on your local repository (Your repository). For that, you have to click the fork button on the repo of the existing repository on GitHub.

- **What is forking:** Forking any repository means make a copy of a real repository in your GitHub account and make changes in your copy. Thus, a real repository won't get affected by your code changes. (After that you have to make a pull request to the real repository for merging your code change, we will come to that part later)
- **How to do fork:** Just go to the real repo and tap on the fork button



**Copy URL:** Then a copy of real repository will be created in your local repository. After that, you have to copy the URL from your local repo. For doing that click to code and copy the URL



After that, you have to create a file on your desktop. Then open Git Bash and go to the file using `cd` command and click enter and type `git clone <copied url>` to copy the code in your desktop file. With that, you are able to get the code on your desktop.

Usage: `git clone [url]`

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

```
$ git init
Initialized empty Git repository in /home/dell/new-folder/.git/
```

## git add:

This command updates the index using the current content found in the working tree and then prepares the content in the staging area for the next commit.

Usage: `git add [file]`

This command adds a file to the staging area.

Usage: `git add *`

This command adds one or more to the staging area.

```
$ git add my_new_file.txt
```

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
    new file:   my_new_file.txt
```



```
$ git add .
```

```
$ git status
```

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

```
new file: another_file.js
```

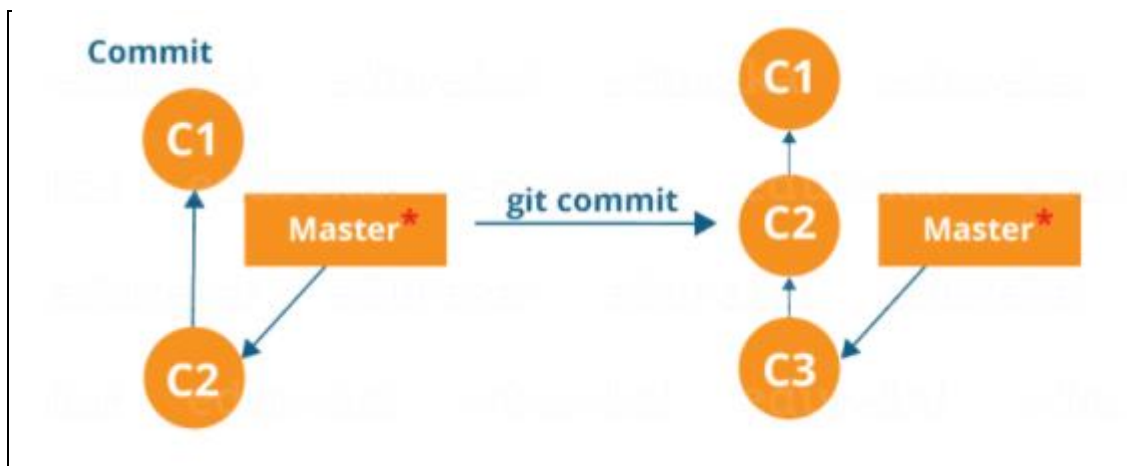
```
new file: my_file.ts
```

```
new file: my_new_file.txt
```

```
new file: new_file.rb
```

## git commit:

It refers to recording snapshots of the repository at a given time. Committed snapshots will never change unless done explicitly. Let me explain how commit works with the diagram below:



Here, C1 is the initial commit, i.e. the snapshot of the first change from which another snapshot is created with changes named C2. Note that the master points to the latest commit.

Now, when I commit again, another snapshot C3 is created and now the master points to C3 instead of C2.

Git aims to keep commits as lightweight as possible. So, it doesn't blindly copy the entire directory every time you commit; it includes commit as a set of changes, or "delta" from one version of the repository to the other. In easy words, it only copies the changes made in the repository.

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 -m "Add three files"
```

```
[master (root-commit) abfbdeb] Add three files
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 another_file.js
create mode 100644 my_new_file.txt
create mode 100644 new_file.rb
```

**Amend commits:** In order to edit the very last commit, one that head points too, it's very difficult to edit older commits because it violates the data integrity.

```
$ git add file-i-forgot-to-add.html
$ git commit --amend -m "Add the remaining file"
```

## git diff:

When working with Git, it is quite common **to use different branches** in order to have work clearly separated from the main codebase.

However, when working on those branches, you might want to merge branches in order to have the resulting work in your main branch.

Before merging, you already know that **you have to compare the differences between the two branches**.

**Comparing two branches** is very beneficial: it can be used as a quick way to see if you will have merging conflicts.

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 diff --color-words new_feature..master -- one line diff`

`git diff --color-words new_feature..master^ -- previous commit of master`

```
$ git diff master..feature
```

```
diff --git a/file-feature b/file-feature
new file mode 100644
index 0000000..add9a1c
--- /dev/null
+++ b/file-feature
@@ -0,0 +1 @@
+this is a feature file
```

```
$ git diff master..feature -- <file>
```

```
$ git diff master..feature -- README
```

```
diff --git a/README b/README
new file mode 100644
index 0000000..add9a1c
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+this is the README file
```

```
$ git diff --oneline --graph <branch>..<current_branch>
```

```
* 391172d (HEAD -> <current_branch>) Commit 2
* 87c800f Commit 1
```

```
$ git diff --oneline --graph master..feature
```

```
* 391172d (HEAD -> feature) My feature commit 2
* 87c800f My feature commit 1
```

## git revert:

Revert - changes will take all of the changes and flip them around, anything that was deleted will be added again and anything that was modified will be in previous state. It will be complete mirror image of that commit. For undo a commit completely and totally, we can use revert.

Usage: `git revert`

Usage: `git revert commit_id`

This command does the automatic revert on commit.

Usage: `git revert -n commit_id` (master|reverting)

Usage: `git revert --abort`

Usage: `git revert --continue`

In order to revert the last Git commit, use the “git revert” and specify the commit to be reverted which is “HEAD” for the last commit of your history.

```
$ git revert HEAD
```

The “git revert” command is slightly different from the “git reset” command because **it will record a new commit with the changes introduced by reverting the last commit.**

Note also that with “git reset” you specified “HEAD~1” because the reset command sets a new HEAD position while reverting actually reverts the commit specified.

As a consequence, you will have to commit the changes again for the files to be reverted and for the commit to be undone.

As a consequence, let’s say that you have committed a new file to your Git repository but you want to revert this commit.

```
$ git log --oneline --graph

* b734307 (HEAD -> master) Added a new file named "file1"
* 90f8bb1 Second commit
* 7083e29 Initial repository commit
```

When executing the “git revert” command, Git will automatically open your text editor in order to commit the changes.

```
Revert "Added a new file named file1"

This reverts commit 1fa26e9184117d6a5a6d29e61f3ef256cf114e45.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is ahead of 'origin/master' by 8 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   deleted:    file1
#
```

When you are done with the commit message, a message will be displayed with the new commit hash.

```
[master 2d40a2c] Revert "Added a new file named file1"
1 file changed, 1 deletion(-)
delete mode 100644 file1
```

Now if you were to inspect your Git history again, you would notice that a new commit was added in order to undo the last commit from your repository

```
$ git log --oneline --graph

* 2d40a2c (HEAD -> master) Revert "Added a new file named file1"
* 1fa26e9 Added a new file named file1
* ee8b133 Second commit
* a3bdedf Initial commit
```

## git reset:

git reset is used for undo the commits. It allows us to specify where the HEAD pointer should point too. But normally we let the git manage the head pointer for us. we make a commit and head pointer to move to that commit. Again we make a commit git will move the head to that commit and point there. But HERE we are saying I want to move the head here as required and that where you are going to do recording & start making our commits.

Eg: - HEAD pointer is like play and rewind law on tape recorder. We are recording an audio in tape recorder and you stop, that's our commit where head points too, If you start recording again it will start from where you have stopped. What if we want to rewind 10 mins back and then record.

Git reset does the same thing, it says lets rewind back to our previous commit and that's where we gonna record our audio and we gonna override whatever came after that. GIT reset allows moves the pointer that one thing it does in every case

**Soft reset --** moves head pointer and does nothing else -- not gonna change the staging index and working directory It's the most safest reset that's why it's called soft.

**Mixed reset --** our working directory contains all the changes, we haven't lost it. We can add it and commit it.. Staging index looks like same as repository..

**Hard reset --** changes that came after that commit are completely gone, they doesn't exist in staging nor working dir nor matched repo. use this with cautious when have gone wrong and everything you want to reset completely to specify point after that commit...

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.

Usage: `git reset --hard [commit]`

This command discards all history and goes back to the specified commit.

In order to undo the last commit and discard all changes in the working directory and index, execute the “git reset” command with the “--hard” option and specify the commit before HEAD (“HEAD~1”).

```
$ git reset --hard HEAD~1
```

Be careful when using “--hard” : changes will be removed from the working directory and from the index, you will lose all modifications.

Back to the example we have detailed before, let’s say that you have committed a new file to your Git repository named “file1”.

```
$ git log --oneline --graph

* b734307 (HEAD -> master) Added a new file named "file1"
* 90f8bb1 Second commit
* 7083e29 Initial repository commit
```

Now, let’s pretend that you want to undo the last commit and discard all modifications.

```
$ git reset --hard HEAD~1

HEAD is now at 90f8bb1 Second commit
```

Let’s now see the state of our Git repository.

```
$ git status

On branch master
Your branch is up to date with origin/master
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Usage: `git reset --soft [commit]`

This command does not discards working and staging area

The easiest way to undo the last Git commit is to execute the “git reset” command with the “--soft” option that will preserve changes done to your files. You have to specify the commit to undo which is “HEAD~1” in this case.

The last commit will be removed from your Git history.

```
$ git reset --soft HEAD~1
```

If you are not familiar with this notation, “HEAD~1” means that you want to reset the HEAD (the last commit) to one commit before in the log history.

```
$ git log --oneline

3fad532 Last commit (HEAD)
3bnaj03 Commit before HEAD (HEAD~1)
vcn3ed5 Two commits before HEAD (HEAD~2)
```

The “[git reset](#)” command can be seen as the **opposite of the “git add”** command, essentially adding files to the Git index.

When specifying the “--soft” option, Git is instructed not to modify the files in the working directory or in the index at all.

As an example, let’s say that you have added two files in your most recent commit but you want to perform some modifications on this file.

```
$ git log --oneline --graph

* b734307 (HEAD -> master) Added a new file named "file1"
* 90f8bb1 Second commit
* 7083e29 Initial repository commit
```

As a consequence, you will use “git reset” with the “--soft” option in order **to undo the last commit** and perform additional modifications.

```
$ git reset --soft HEAD~1

$ git status

On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file1

$ git log --oneline --graph

* 90f8bb1 (HEAD -> master) Second commit
* 7083e29 Initial repository commit
```

As you can see, by undoing the last commit, the file is still in the index (changes to be committed) but the commit was removed.

Usage: `git reset --mixed [commit]`

This command keeps all changes in working Directory.

In order to undo the last Git commit, keep changes in the working directory but NOT in the index, you have to use the “git reset” command with the “--mixed” option. Next to this command, simply append “HEAD~1” for the last commit.

```
$ git reset --mixed HEAD~1
```

As an example, let’s say that we have added a file named “file1” in a commit that we need to undo.

```
$ git log --oneline --graph
* b734307 (HEAD -> master) Added a new file named "file1"
* 90f8bb1 Second commit
* 7083e29 Initial repository commit
```

To undo the last commit, we simply execute the “git reset” command with the “--mixed” option.

```
$ git reset --mixed HEAD~1
```

When specifying the “--mixed” option, the file will be removed from the Git index but not from the working directory.

As a consequence, the “--mixed” is a “mix” between the soft and the hard reset, hence its name.

```
$ git status

On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file1

nothing added to commit but untracked files present (use "git add" to track)
```



## git status:

The **git status** command lists all the modified files which are ready to be added to the local repository.

Usage: **git status**

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

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

## git log:

Git log command is one of the most usual commands of git. It is the most useful command for Git. Every time you need to check the history, you have to use the git log command. The basic git log command will display the most recent commits and the status of the head. It will use as:

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.

Usage: **git log --oneline**

This command lists version history in oneline ignoring other details like author, date, email etc.

Usage: **git log -n(digit)[numeric Digit]**

This command lists version history from latest commit.

## git show:

Usage: **git show [commit]**

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

```
$ git show 02fda84
commit 02fda8411c09d95a96625704ccc98b13fe44788a
Author: Sagargoud <goud.sagar.t@gmail.com>
Date: Sat Nov 12 15:54:15 2016 +0530

    new files

diff --git a/express.html b/express.html
index 41001ab..67a0d15 100644
--- a/express.html
+++ b/express.html
@@ -1,2 @@
<h1> head </head>
-
+ffffffffffffffff
diff --git a/index.html b/index.html
index 2e3c6f6..21c6f0d 100644
--- a/index.html
+++ b/index.html
@@ -1,2 @@
-hello world
+hello world
```

## git tag:

On Git, **tags** are used in order to define commits in your history that may be more important than others.

When you are performing a merge commit, right before deploying, you might want **to tag this commit**.

This way, if you choose to go back to the previous version, you will be able to find the commit in the blink of an eye.

However, in order to find this commit, you will need **to list the existing Git tags of your repository**

Usage: `git tag [commitID]`

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

### List Local Git Tags

```
$ git tag  
  
v1.0  
v2.0
```

Execute “git tag” with the “-n” option in order to have an extensive description of your tag list.

```
$ git tag -n
```

Specify a tag pattern with the “-l” option followed by the tag pattern.

```
$ git tag -l <pattern>
```

### List Remote Git Tags

```
$ git ls-remote --tags <remote>
```

```
$ git ls-remote --tags origin
```

```
53a7dcf1ca57e05d456321b406730b39dc8ed75e      refs/tags/v1.0  
7a9ad7fd794bf52a11de43aacc6010978e6100d3      refs/tags/v2.0
```

### Fetching Remote Tags Easily

To fetch tags from your remote repository, use “[git fetch](#)” with the “-all” and the “-tags” options.

```
$ git fetch --all --tags

Fetching origin
From git-repository
   53a7dc..7a9ad7   master    -> origin/master
* [new tag]         v1.0      -> v1.0
* [new tag]         v1.0      -> v2.0
```

## git branch:

Branches are most powerful feature in Git. In git branches are cheap, we mean that it won't lot of headaches, no more process power, storage, space is needed, easy to create, delete and work with different branches which allows us to try new ideas. Suppose, we are on master branch and we got an idea which is not working out ,instead of making lot of commits to master branch and trying to undo those and if those don't work out, instead create branch try out new ideas.. if wont work out throw away the branch. If those changes work u can fold those chnages to master this process we called merging.

This branching feature is very important when we are collaborating with others.

Eg: user feedback form -- we can add this feature in web page by creating a new branch and poeple can work on master branch. Once the feedback form is done we can collabrate with others and merge it to master. Git will swap out the all the changes of user feedback and when u switvh back to master al changes are gone.

Master branch -- we have four commit, we decide revising the navigation and we are not sure it will work out or not. So we create new branch and we have all the changes in our wokring directory.. We comit those to new branch

Now all user features are on new branch... if any one checkouts master they wont see the user feedback changes. The movement u switvh the branch.. the head points where the branch is checkedout

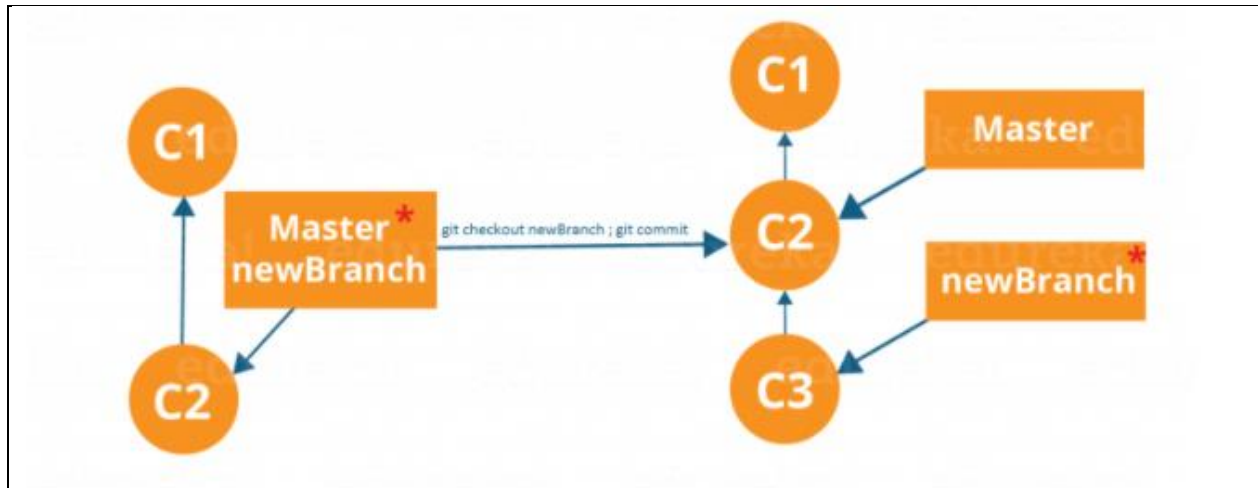
**git branch <branch-name>**



The diagram above shows the workflow when a new branch is created. When we create a new branch it originates from the master branch itself.

Since there is no storage/memory overhead with making many branches, it is easier to logically divide up your work rather than have big chunky branches.

Now, let us see how to commit using branches.



Branching includes the work of a particular commit along with all parent commits. As you can see in the diagram above, the newBranch has detached itself from the master and hence will create a different path.

Usage: `git branch`

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

Usage: `git branch [branch name]`

This command creates branch.

Usage: `git branch -d [branch name]`

```
$ git branch -d release

Deleted branch feature (was bd6903f).
```

This command deletes the branch.

Usage: `git branch -D [branch name]`

```
$ git branch -d <branch>

error: The branch 'branch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D branch'.

$ git branch -D <branch>
Deleted branch feature (was 022519a).
```

This command deletes the branch forcefully.

Usage: `git branch -m [old branch] [new branch]`

This command renames the branch.

Usage: `git branch --merged`

This command tells which branch has all commits of master branch and which other branches can be deleted.

```
#Git branch --merged
master
new_feature
* shorten_title
shorten_title - contains all features of maser and new_feature. we can delete new_feature
```

### How to set and display branch name on Linux Prompt:

```
__git_ps1 function to show the branch on prompts.
#echo $PS1
Edit bashrc or bash.profile and search below and replace. Finally Source the file.
export PS1='(__git_ps1 "%s") > '

export PS1="\W$__git_ps1 "%s") > '

source ~/.bash.profile
```

## git checkout:

Usage: `git checkout [branch name]`

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

```
$ git checkout master
*master
user-profile
```

Usage: `git checkout -b [branch name]`

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

```
$ git checkout -b admin-panel  
Switched to branch "admin-panel"
```

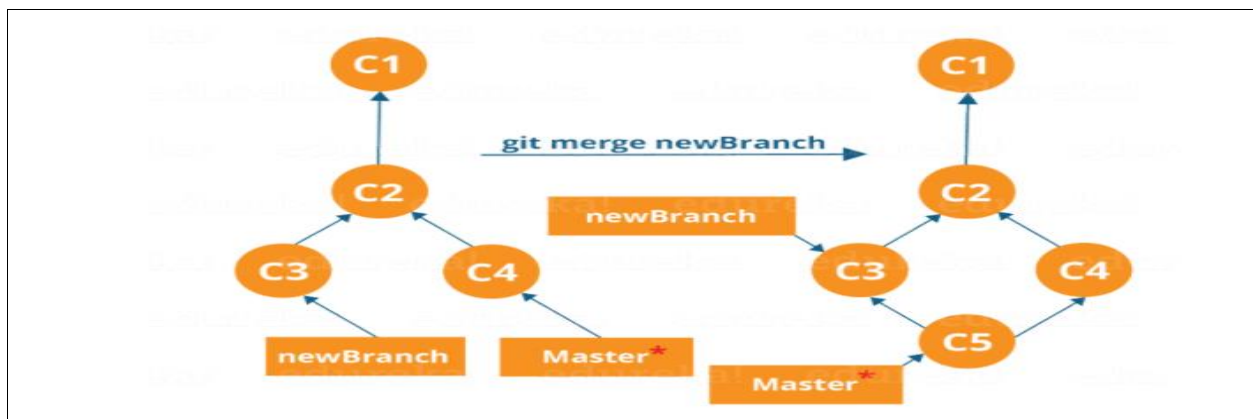
## git merge:

Merging is the way to combine the work of different branches together. This will allow us to branch off, develop a new feature, and then combine it back in.

Usage: `git merge [branch name]`

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

Eg: The diagram below shows us two different branches-> new Branch and master. Now, when we merge the work of new Branch into master, it creates a new commit which contains all the work of master and new Branch

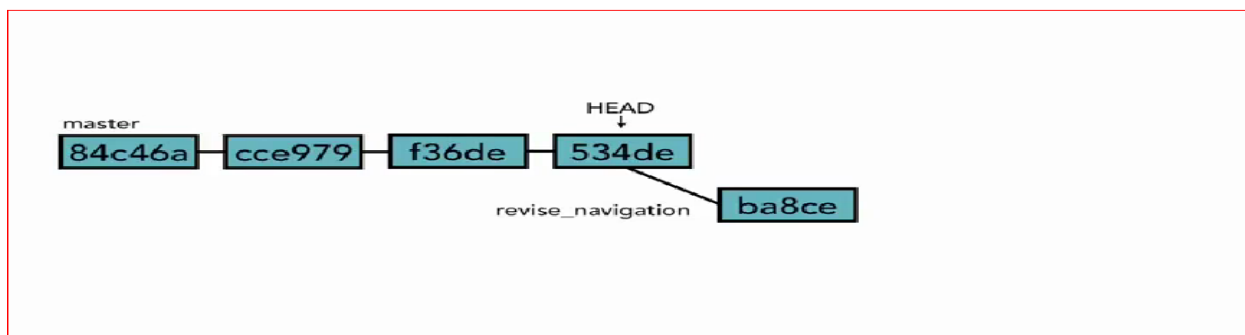


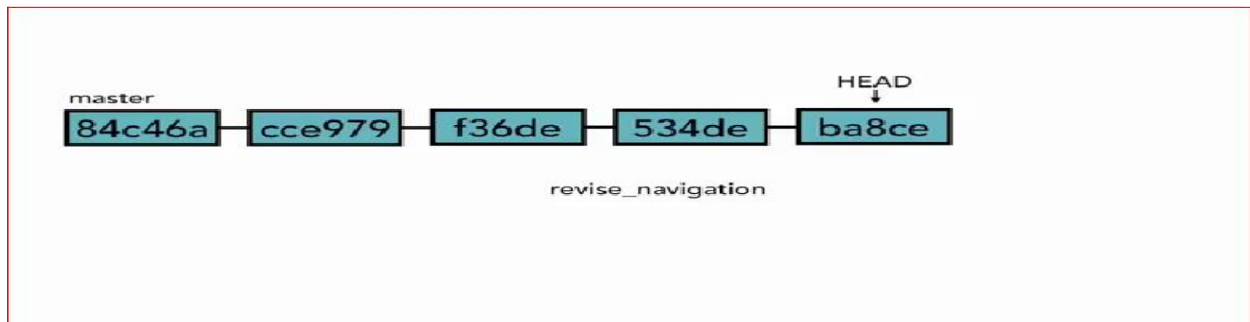
Now let us merge the two branches with the command below:

```
git merge <branch_name>
```

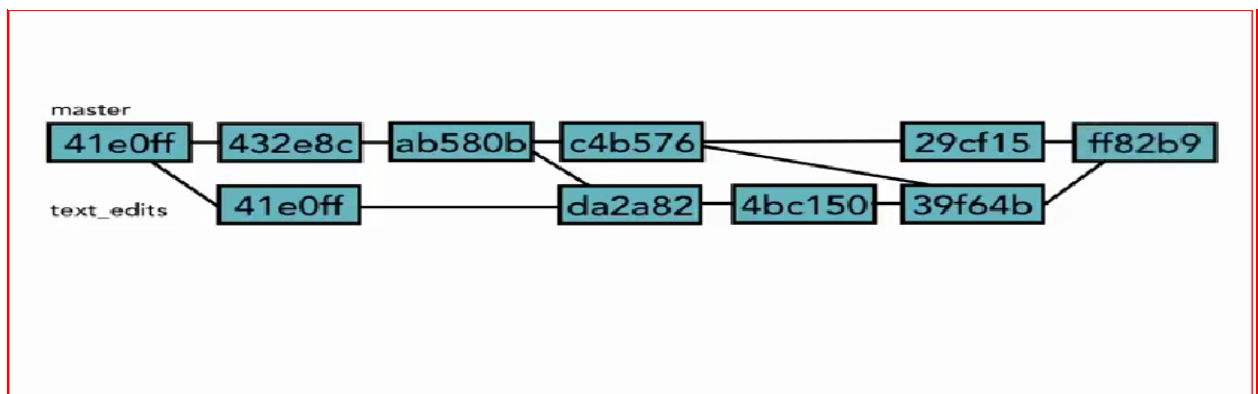
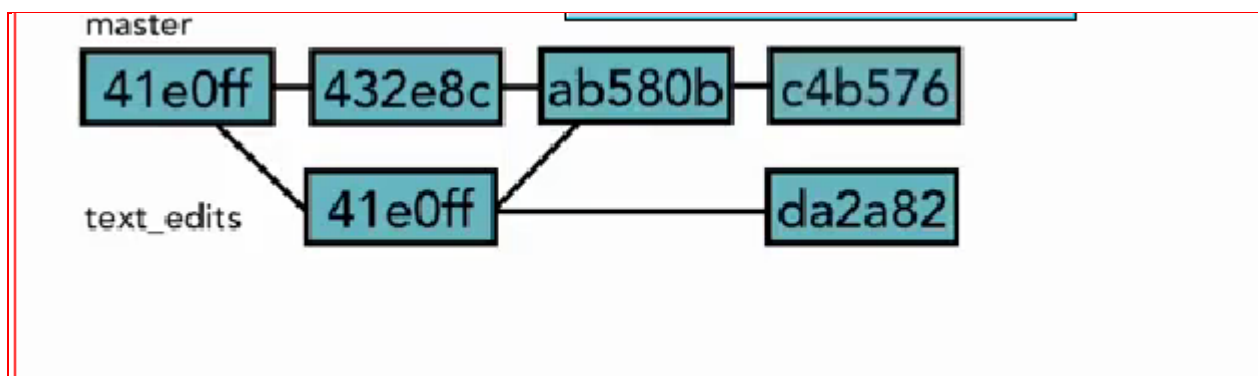
It is important to know that the branch name in the above command should be the branch you want to merge into the branch you are currently checking out. So, make sure that you are checked out in the destination branch.

## FAST FORWARD MERGE:





## NON FAST FORWARD MERGE:



## Merge a file from one branch to another:

Sometimes, you may want to merge the content of a specific file in one branch into another. For example, you want to merge the content of a file `index.html` in the master branch of a project into the development branch. How can you do that?

First, checkout to the right branch (the branch you want to merge the file) if you've not already done so. In our case, it's the development branch.

```
git checkout development
```

Then merge the file using the checkout --patch command.

#### **git checkout --patch master index.html**

If you want to completely overwrite the index.html file on the development branch with the one on the master branch, you leave out the --patch flag.

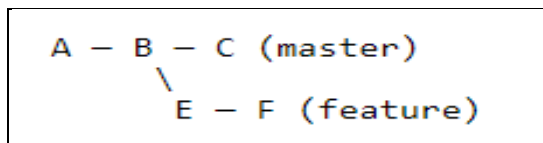
#### **git checkout master index.html**

Also, leave out the --patch flag if the index.html file does not exist on the development branch.

## **Scenario: Merge a file from one branch to another:**

### **1. Creating Master and Feature Branches**

Here is the scenario we will create:



In the above example, we are taking the following path:

- Commit A: we add a.txt file in the 'master' branch
- Commit B: we add b.txt file in the 'master' branch
- At this stage, we create the branch 'feature' which means it will have a.txt and b.txt
- Commit C: we add c.txt file in the 'master' branch
- We go to the 'feature' branch
- Commit E: we modify a.txt in 'feature' branch
- Commit F: we modify b.txt in 'feature' branch
- You can create a folder and run the following code inside the folder to create the above situation:



```
git init
touch a.txt
git add -A
git commit -m "Commit A: added a.txt"

touch b.txt
git add -A
git commit -m "Commit B: added b.txt"
git branch feature

touch c.txt
git add -A
git commit -m "Commit C: added c.txt"
git status
git checkout feature

echo aaa > a.txt
git add -A
git commit -m "Commit E: modified a.txt"

echo bbb > b.txt
git add -A
git commit -m "Commit F: modified b.txt"
```

## 1. Simple Merge

Let's use the log command to check both branches.

### Results for 'master':

```
$ git checkout master
Switched to branch 'master'

$ git log --oneline
2bbde47 Commit C: added c.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt  b.txt  c.txt
```

### Results for 'feature':

```
$ git checkout feature
Switched to branch 'feature'

$ git log --oneline
0286690 Commit F: modified b.txt
7c5c85e Commit E: modified a.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt  b.txt
```

Notice how the feature branch does not have Commit C

Now let's run merge 'feature' branch with 'master' branch. You will be asked to enter a comment. In the comment, add "Commit G:" at the beginning to make it easier to track.

```
$ git checkout master
Switched to branch 'master'

$ git merge feature
Merge made by the 'recursive' strategy.
a.txt | 1 +
b.txt | 1 +
2 files changed, 2 insertions(+)
```

Results for 'master':

```
$ git checkout master
Already on 'master'

$ git log --oneline
d086ff9 Commit G: Merge branch 'feature'
0286690 Commit F: modified b.txt
7c5c85e Commit E: modified a.txt
2bbde47 Commit C: added c.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt b.txt c.txt
```

Results for 'feature':

```
$ git checkout feature
Switched to branch 'feature'

$ git log --oneline
0286690 Commit F: modified b.txt
7c5c85e Commit E: modified a.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt  b.txt
```

In the 'master' branch, you will notice there is a new commit G that has merged the changes from 'feature' branch. Basically, the following action has taken place:

```
A - B - C - G (master)
      \   /
       E - F (feature)
```

In the Commit G, all the changes from 'feature' branch have been brought into the master branch. But the 'feature' branch itself has remained untouched due to the merge process. Notice the hash of each commit. After the merge, E (7c5c85e) and F (0286690) commit has the same hash on the 'feature' and 'master' branch.

```
$ git checkout master
Switched to branch 'master'

$ git merge feature
Merge made by the 'recursive' strategy.
a.txt | 1 +
b.txt | 1 +
2 files changed, 2 insertions(+)
```

#### Results for 'master':

```
$ git checkout master
Already on 'master'

$ git log --oneline
d086ff9 Commit G: Merge branch 'feature'
0286690 Commit F: modified b.txt
7c5c85e Commit E: modified a.txt
2bbde47 Commit C: added c.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt b.txt c.txt
```

#### Results for 'feature':

```
$ git checkout feature
Switched to branch 'feature'

$ git log --oneline
0286690 Commit F: modified b.txt
7c5c85e Commit E: modified a.txt
b430ab5 Commit B: added b.txt
6f30e95 Commit A: added a.txt

$ ls
a.txt  b.txt
```

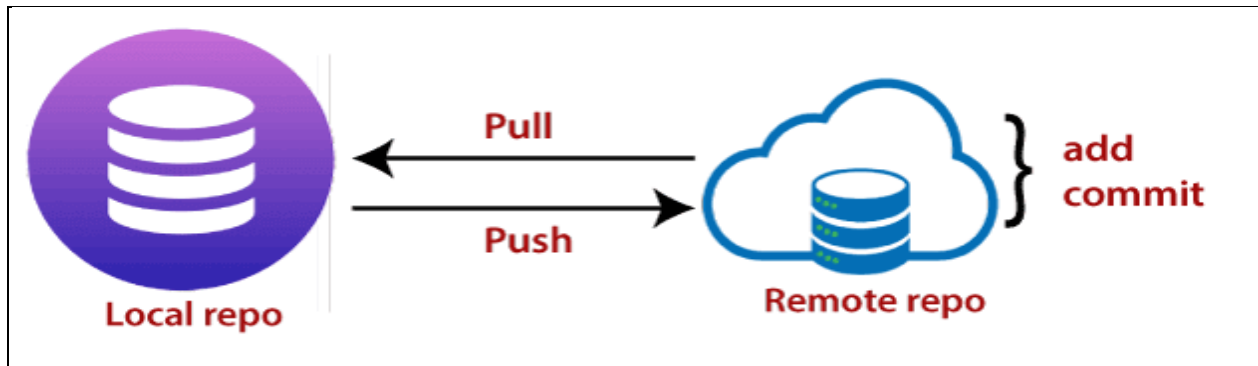
## git push:

This command transfers commits from your local repository to your remote repository. It is the opposite of pull operation.

Pulling imports commits to local repositories whereas pushing exports commits to the remote repositories.

The use of **git push** is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.



### **git push <remote>**

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.

```
C:\git-rename (quickfix -> origin)
λ git push origin -u quickfix
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 243 bytes | 243.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To
  b0d1562..199f253  quickfix -> quickfix
Branch 'quickfix' set up to track remote branch 'quickfix' from 'origin'.
```

## git pull:

The **git pull** command fetches changes from a remote repository to a local repository. It merges upstream changes in your local repository, which is a common task in Git based collaborations.

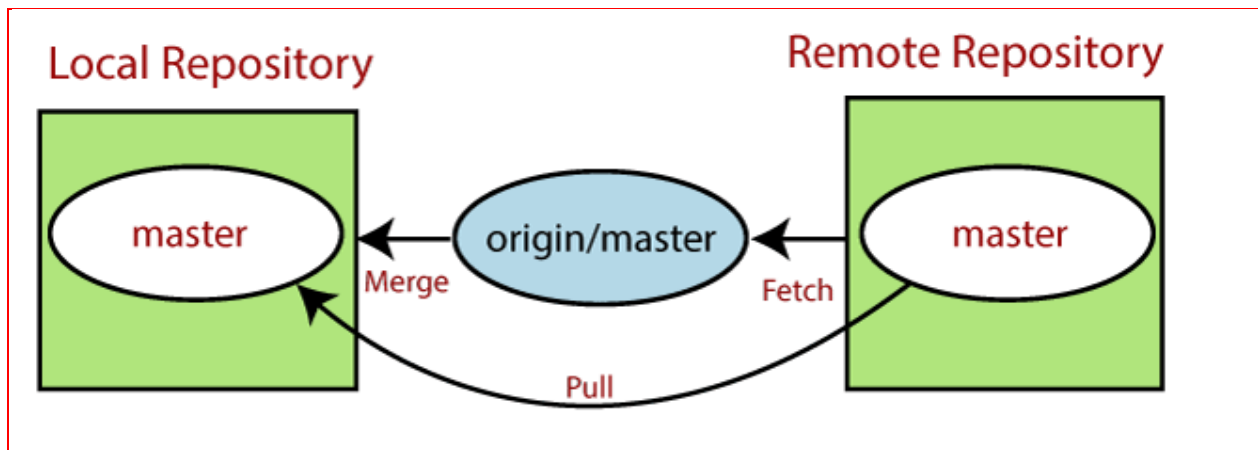
The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repository.

### git pull origin master

This command will copy all the files from the master branch of remote repository to your local repository.

Usage: `git pull [Repository Link]`

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



```
HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$ git pull origin master
From https://github.com/ImDwivedi1/GitExample2
 * branch          master      -> FETCH_HEAD
updating 0a1a475..828b962
Fast-forward
 design2.css | 1 +
 1 file changed, 1 insertion(+)

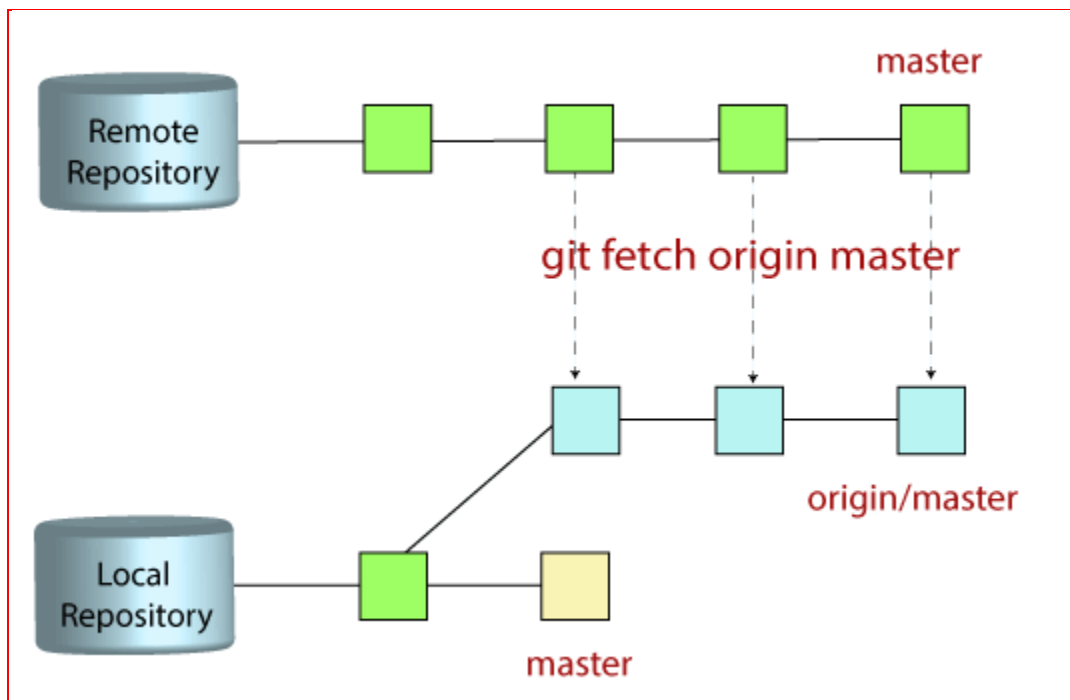
HiManshu@HiManshu-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$
```

## git fetch:

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.

Usage: `git fetch`

This command fetches all changes from remote to origin/master.



The **"git fetch" command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

**Eg:**

```
MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git fetch upstream master
remote: warning: multi-pack bitmap is missing required reverse index
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 630 bytes | 3.00 KiB/s, done.
From https://github.com/GoudSagar/Hello_World12
 * branch      master       -> FETCH_HEAD
 * 0988377..53d7b1a master   -> upstream/master

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git log -n1
commit 5a11b4405d8f641ed133c9944bee41c9569d88a1 (HEAD -> master)
Merge: b09b7a7 0988377
Author: sagargoud009 <meeeeet.sagar@gmail.com>
Date:   Wed Apr 14 17:34:40 2021 +0530

    Merge branch 'master' of https://github.com/GoudSagar/Hello_World12

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git merge upstream/master
Merge made by the 'recursive' strategy.
 x7 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 x7

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git log -n1
commit d2a9e367b5543ced41baadcd2d021faff9975f42 (HEAD -> master)
Merge: 5a11b44 53d7b1a
Author: sagargoud009 <meeeeet.sagar@gmail.com>
Date:   Wed Apr 14 17:43:28 2021 +0530

    Merge remote-tracking branch 'upstream/master'

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
```

## Scenario 1: To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch https://github.com/ImDwivedi1/Git-Example.git
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            HEAD       -> FETCH_HEAD

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ |
```

## Scenario 2: To fetch a specific branch:

We can fetch a specific branch from a repository. It will only access the element from a specific branch

```
$ git fetch <branch URL><branch name>
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch https://github.com/ImDwivedi1/Git-Example.git Test
warning: no common commits
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            Test       -> FETCH_HEAD

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ |
```

## Scenario 3: To fetch all the branches simultaneously:

The git fetch command allows to fetch all branches simultaneously from a remote repository.

```
$ git fetch -all
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch --all
Fetching origin
From https://github.com/ImDwivedi1/Git-Example
 * [new branch]      master     -> origin/master
 * [new branch]      Test       -> origin/Test

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$
```

## Scenario 4: To synchronize the local repository:

Suppose, your team member has added some new features to your remote repository. So, to add these updates to your local repository, use the git fetch command. It is used as follows.

```
$ git fetch origin
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
* [new branch]      test2      -> origin/test2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-Example (master)
$ |
```

In the above output, new features of the remote repository have updated to my local system. In this output, the branch **test2** and its objects are added to the local repository.

The git fetch can fetch from either a single named repository or URL or from several repositories at once. It can be considered as the safe version of the git pull commands.

The git fetch downloads the remote content but not update your local repo's working state. When no remote server is specified, by default, it will fetch the origin remote.

## Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

```
git pull = git fetch + git merge
```



## Git Fetch vs. Pull

Some of the key differences between both of these commands are as follows:

git fetch	git pull
Fetch downloads only new data from a remote repository.	Pull is used to update your current HEAD branch with the latest changes from the remote server.
Fetch is used to get a new view of all the things that happened in a remote repository.	Pull downloads new data and directly integrates it into your current working copy files.
Fetch never manipulates or spoils data.	Pull downloads the data and integrates it with the current working file.
It protects your code from merge conflict.	In git pull, there are more chances to create the <b>merge conflict</b> .
It is better to use git fetch command with git merge command on a pulled repository.	It is not an excellent choice to use git pull if you already pulled any repository.

## git stash:

The **git stash** command is probably one of the most powerful commands in Git.

**Git stash** is used in order to **save all the changes done to the current working directory** and to go back to the last commit done on the branch (also called HEAD).

Stashing changes comes with a special set of Git commands designed to **create, delete and apply stashes** at will.

Usage: **git stash save**

This command temporarily stores all the modified tracked files.

```
explore_california(master) > pwd
/Users/kevinskoglund/Documents/explore_california
explore_california(master) > git branch
* master
  seo_title
  shorten_title
  text_edits
explore_california(master) > git checkout shorten_title
Switched to branch 'shorten_title'
explore_california(shorten_title) > git branch --merged
  seo_title
* shorten_title
explore_california(shorten_title) > _
```

```
explore_california(shorten_title) > git status
# On branch shorten_title
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   mission.html
#
no changes added to commit (use "git add" and/or "git commit -a")
explore_california(shorten_title) > git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    mission.html
Please, commit your changes or stash them before you can switch branches.
Aborting
explore_california(shorten_title) > _
```

```
explore_california(shorten_title) > git status
# On branch shorten_title
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   mission.html
#
no changes added to commit (use "git add" and/or "git commit -a")
explore_california(shorten_title) > git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    mission.html
Please, commit your changes or stash them before you can switch branches.
Aborting
explore_california(shorten_title) > git stash save "changed mission page title"
Saved working directory and index state On shorten_title: changed mission page title
HEAD is now at c091faf Swap out - for : in index.html title
explore_california(shorten_title) > git status_
```

Usage: `git stash pop`

This command restores the most recently stashed files.

```
explore_california(master) > pwd
/Users/kevinskoglund/Documents/explore_california
explore_california(master) > git status
# On branch master
nothing to commit (working directory clean)
explore_california(master) > git stash list
stash@{0}: On shorten_title: changed mission page title
explore_california(master) > git stash pop stash@{0}
Auto-merging mission.html
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   mission.html
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{0} (f5a1464fc7aad3c76c77f2e4fd397ac4a825ca59)
explore_california(master) > git stash list
explore_california(master) > _
```

Usage: `git stash list`

This command lists all stashed changesets.

```
explore_california(shorten_title) > pwd
/Users/kevinskoglund/Documents/explore_california
explore_california(shorten_title) > git status
# On branch shorten_title
nothing to commit (working directory clean)
explore_california(shorten_title) > git stash list
stash@{0}: On shorten_title: changed mission page title
```

Usage: `git stash drop`

This command discards the most recently stashed changeset

```
explore_california(shorten_title) > pwd
/Users/kevinskoglund/Documents/explore_california
explore_california(shorten_title) > git status
# On branch shorten_title
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   mission.html
#
no changes added to commit (use "git add" and/or "git commit -a")
explore_california(shorten_title) > git stash list
stash@{0}: On master: change to mission page title
explore_california(shorten_title) > git stash drop stash@{0}
Dropped stash@{0} (bf2b3b6ac42eb43d3f1e571ca19625432bd1b862)
explore_california(shorten_title) > git stash list
explore_california(shorten_title) > _
```

Usage: `git stash apply`

This command brings out the most recent change set and preserving copy in stash.

```
explore_california(shorten_title) > git status
# On branch shorten_title
nothing to commit (working directory clean)
explore_california(shorten_title) > git stash list
stash@{0}: On master: change to mission page title
explore_california(shorten_title) > git stash apply
Auto-merging mission.html
# On branch shorten_title
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   [REDACTED]
#
no changes added to commit (use "git add" and/or "git commit -a")
explore_california(shorten_title) > git stash list
stash@{0}: On master: change to mission page title
explore_california(shorten_title) > _
```

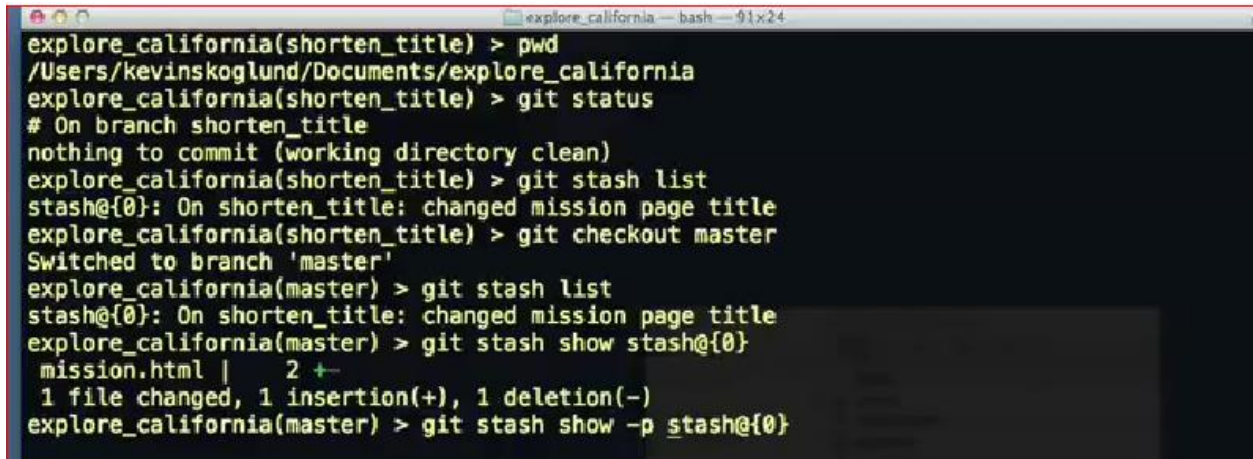


Usage: `git stash clear`

This command **deletes** all the Git stashes in your stack

Usage: `git stash show`

This command **shows** git stashes in your stack.

A terminal window titled 'explore\_california -- bash -- 91x24' showing a series of git commands and their outputs. The user is in a directory '/Users/kevinskoglund/Documents/explore\_california'. They run 'git status' and see they are on the 'shorten\_title' branch with no changes. Then they run 'git stash list' and see a stash with changes to 'mission page title'. They run 'git checkout master' and switch to the 'master' branch. Then they run 'git stash list' again and see the same stash. They run 'git stash show stash@{0}' and see a diff for 'mission.html' with 2 lines added and 1 line deleted. Finally, they run 'git stash show -p stash@{0}' to see the full patch.

```
explore_california(shorten_title) > pwd
/Users/kevinskoglund/Documents/explore_california
explore_california(shorten_title) > git status
# On branch shorten_title
nothing to commit (working directory clean)
explore_california(shorten_title) > git stash list
stash@{0}: On shorten_title: changed mission page title
explore_california(shorten_title) > git checkout master
Switched to branch 'master'
explore_california(master) > git stash list
stash@{0}: On shorten_title: changed mission page title
explore_california(master) > git stash show stash@{0}
mission.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
explore_california(master) > git stash show -p stash@{0}
```

## git cherry-pick:

**git cherry-pick** is a powerful **command** that enables arbitrary **Git** commits to be **picked** by reference and appended to the current working HEAD. **Cherry picking** is the act of **picking** a commit from a branch and applying it to another. **git cherry-pick** can be useful for undoing changes

Usage: `git cherry-pick`

This command temporarily stores all the modified tracked files.

## Cherry-pick using Git commit hash

The easiest way to cherry-pick a commit is to use the “[cherry-pick](#)” command with the commit hash.

```
$ git cherry-pick <hash>
```

In order to cherry-pick changes, you will need to identify your commit hashes.

In order to see the commit hashes for your current branch, simply run the “git log” command with the “--oneline” option in order to make it more readable.

```
$ git log --oneline

45ab1a8 (HEAD -> branch2) added gitignore
808b598 (branch) Initial commit
```

By default, the log command will display the commits from the history beginning until the top of your current branch.

As a consequence, you may not see commits that are not related to your current branch timeline.

If you want to see commits related to a specific branch, **specify the branch name when running the “git log” command.**

```
$ git log --oneline master

93ae442 (master) committed changes
44ee0d4 added gitignore
808b598 (branch) Initial commit
```

As you can see, one additional commit was displayed: you can now use this hash in order to cherry-pick your commit.

```
$ git cherry-pick 93ae442

[master 299a73d] added file
Date: Wed Nov 20 16:04:52 2019 -0500
1 file changed, 1 insertion(+)
create mode 100644 file.txt
```

## Cherry-pick from another branch

In order to pick commits from another branch, **you need to list commits that were performed on this other branch using the “git log” command.**

```
$ git log --oneline <branch>
```

Let's say for example that I want to cherry-pick a commit from the feature branch.

```
$ git log --oneline feature

93ae442 (master) Feature 1
44ee0d4 Feature 2
808b598 (branch) Initial commit
```

Now, you can go to the branch where you want the commit to be cherry-picked, let's call it “master” in this case.

```
$ git checkout master

$ git cherry-pick 93ae442

[master 299a73d] added file
Date: Wed Nov 20 16:04:52 2019 -0500
```

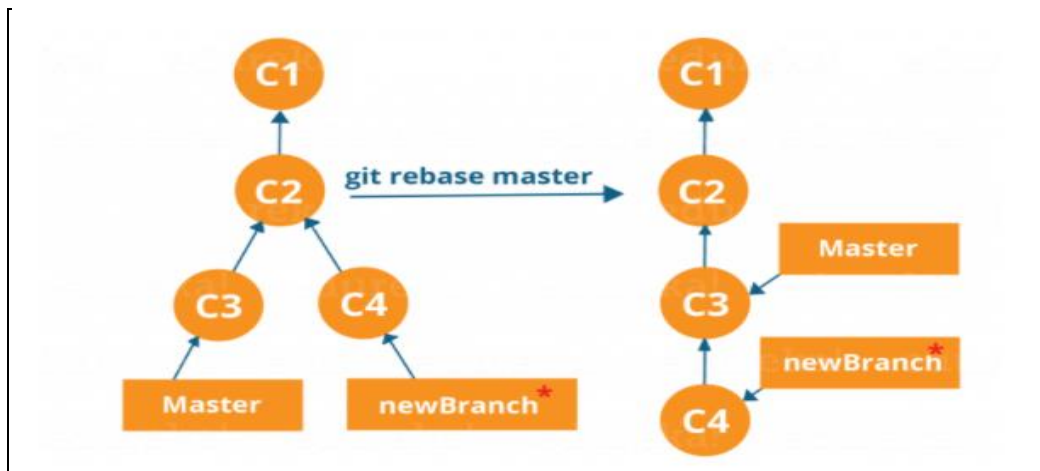
```
1 file changed, 1 insertion(+)  
create mode 100644 file.txt
```

## git rebase:

This is also a way of combining the work between different branches. Rebasing takes a set of commits, copies them and stores them outside your repository.

The advantage of rebasing is that it can be used to make linear sequence of commits. The commit log or history of the repository stays clean if rebasing is done.

- Actually changing the history of commits you are merging..
- Changing the root commit and branches based off.
- Resetting the base commit to most recent commit of the branch u planning to merge.



Now, our work from newBranch is placed right after master and we have a nice linear sequence of commits.

**Note:** *Rebasing also prevents upstream merges, meaning you cannot place master right after newBranch.*

Now, to rebase master, type the command below in your Git Bash:

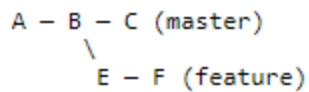
```
git rebase master
```

This command will move all our work from current branch to the master. They look like as if they are developed sequentially, but they are developed parallelly.

## Rebasing from one branch to another

## 1. Creating Master and Feature Branches

Here is the scenario we will create:



In the above example, we are taking the following path:

1. Commit A: we add a.txt file in the 'master' branch
2. Commit B: we add b.txt file in the 'master' branch
3. At this stage, we create the branch 'feature' which means it will have a.txt and b.txt
4. Commit C: we add c.txt file in the 'master' branch
5. We go to the 'feature' branch
6. Commit E: we modify a.txt in 'feature' branch
7. Commit F: we modify b.txt in 'feature' branch

You can create a folder and run the following code inside the folder to create the above situation:

```
git init
touch a.txt
git add -A
git commit -m "Commit A: added a.txt"

touch b.txt
git add -A
git commit -m "Commit B: added b.txt"
git branch feature

touch c.txt
git add -A
git commit -m "Commit C: added c.txt"
git status
git checkout feature

echo aaa > a.txt
git add -A
git commit -m "Commit E: modified a.txt"

echo bbb > b.txt
git add -A
git commit -m "Commit F: modified b.txt"
```

## 2. Simple Rebase

Let's use the log command to check both branches.

Results for 'master':

```
$ git checkout master
Switched to branch 'master'

$ git log --oneline
7f573d8 Commit C: added c.txt
795da3c Commit B: added b.txt
0f4ed5b Commit A: added a.txt

$ ls
a.txt  b.txt  c.txt
```

Results for 'feature':

```
$ git checkout feature
Switched to branch 'feature'

$ git log --oneline
8ed0c4e Commit F: modified b.txt
6e12b57 Commit E: modified a.txt
795da3c Commit B: added b.txt
0f4ed5b Commit A: added a.txt

$ ls
a.txt b.txt
```

Let's rebase from the 'feature' branch.

```
$ git checkout feature
Switched to branch 'feature'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Commit E: modified a.txt
Applying: Commit F: modified b.txt
```

```
A - B - C
      \
      E' - F' (feature, master)
```

Rebasing is a useful tool when you want to clean up the history of your work. However, there is a danger which has given birth to the golden rule.

## git clean:

When working with Git, it is quite usual to **accumulate** many different branches for the different features we are working on.

However, when merged with our master branch, you may want to **clean up unused branches** in order for your Git workspace to be **more organized**.

As a developer, it can be quite tiring to have references to hundreds of different branches in our Git repository.

git clean will remove all the untracked files from the repository and no longer you can recover it...

Usage: `git clean -n`



This command will do the test run and let us know which all files and folder will be cleaned up.

Usage: `git clean -f`

## git rm:

Usage: `git rm [file]`

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

The easiest way to delete a file in your Git repository is to execute the “git rm” command and to specify the file to be deleted

```
$ git rm <file> .  
  
$ git commit -m "Deleted the file from the git repository"  
  
$ git push
```

Eg: Have three files named “file1”, “file2” and “file3” and want to delete the “file1” file from my Git repository.

```
$ git ls-tree -r master
```

By using the “git ls-tree” command, able to see the files tracked on my current branch.

```
C:\git-delete-example (master -> origin)  
λ git ls-tree -r master  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 file1  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 file2  
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 file3
```

In order to delete the file “file1” from the Git repository and from the filesystem, we are going to execute the “git rm” command with the name of the file.

```
$ git rm file1  
rm 'file1'
```

```

C:\git-delete-example (master -> origin)
λ git rm file1
rm 'file1'

C:\git-delete-example (master -> origin)
λ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    file1

```

Executing the “git rm” command, a “deleted” action was added to the changes to be committed.

**It means that the file was removed from the filesystem** but it was not deleted from the index just yet. In order for the changes to be effective, you will have to commit your changes and push them to your remote repository.

```

C:\git-delete-example (master -> origin)
λ git commit -m "Deleted the file1 from the Git repository and filesystem"
[master ebf0c11] Deleted the file1 from the Git repository and filesystem
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 file1

C:\git-delete-example (master -> origin)
λ git push
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 268 bytes | 268.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/      /git-delete-example.git
   c586ed5..ebf0c11  master -> master

```

Now the file should be deleted from the file system and from the index, you can verify it by re-executing the “git ls-tree” command in order to list files in the current index.

```

C:\git-delete-example (master -> origin)
λ git ls-tree -r master
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    file2
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    file3

```

## Delete a branch both locally and remotely

A branch is a version of the repository that is different from the main working project. You may want to read up on Git branches and how to add a branch if you are not familiar with that process.

## How to delete a local branch

To delete a branch locally, make sure you are not on the branch you want to delete. So you have to checkout to a different branch and use the following command:

```
git branch -d <name-of-branch>
```

So if I want to delete a branch named fix/homepage-changes, I'll do the following:

```
git branch -d fix/homepage-changes
```

You can run git branch on your terminal to confirm that the branch has been successfully removed. Sometimes you may have to delete a branch you've already pushed to a remote repository. How can you do this?

## How to delete a remote branch

To delete a branch remotely, you use the following command:

```
git push <remote-name> --delete <name-of-branch>
```

where remote-name is the name of the remote repository you want to delete the branch from. If I want to delete the branch fix/homepage-changes from origin, I'll do this:

```
git push origin --delete fix/homepage-changes
```

The branch will be deleted remotely now.

## Undo a commit

There are times when you've committed your changes incorrectly and you want to undo this commit. Sometimes, you may have even pushed the changes to a remote branch. How do you undo or delete this commit? Let's start with undoing a local commit.

### How to undo a local commit

One way you can undo a commit locally is by using git reset. For example, if you want to undo the last commit made, you can run this command:

```
git reset --soft HEAD~1
```

The --soft flag preserves the changes you've made to the files you committed, only the commit is reverted. However, if you don't want to keep the changes made to the files, you can use the --hard flag instead like this:

```
git reset --hard HEAD~1
```

Note that you should use the --hard flag only when you are sure that you don't need the changes.

Also, note that HEAD~1 points to the last commit. If you want to undo a commit before that, you can use git reflog to get a log of all previous commits. Then use the git reset command with the commit hash (the number you get at the beginning of each line of history). For example, if my commit hash is 9157b6910, I'll do this

```
git reset --soft 9157b6910
```

## How to undo a remote commit

There are times you want to undo a commit you have pushed to a remote repository. You can use git revert to undo it locally and push this change to the remote branch.

First, gets the commit hash using git reflog.

```
git reflog
```

revert it. Assuming my commit hash is 9157b6910, I'll do the following:

```
git revert 9157b6910
```

Finally, push this change to the remote branch.

## git remote:

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

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

Remote repository is Central Clearing House and Remote server just a git repo.

Everything we have done till now is in our local computer for version control and git allows to do it.

Git becomes more powerful when we collaborate with others that are what remotes allow us to do.

When there is a remote server and we can send our changes to remote server so that other people can see it. Others can download the changes to their repositories and upload them back.

We can pull those changes to our repo and work on it. It makes this central repo as central clearance house for all of these different changes that are going on. Remote server is just a git repository.

Git is distributed version control. There is no much diff bet different repos. server and our computer or client no much difference. git server is running

only git software that allows us to communicate with lot of different git clients.

at same time, repository there we store is just a git repo it has branches, commits and head pointers it works as same.

Push -- When u push the changes to remote server creates the same branch with commits with same commit id's. Git also makes the another branch on our local computer that is called origin/master. reference remote server branch and always tries to stay in synch with that.

Fetch -- Commits made in master to get the changes we use fetch... at that time it comes into our origin/master branch

Until we do a merge the changes fetch from remote server wont to master (local) .. it will be in origin/master only.

There are not duplicate copies of objects..git uses pointer.. thats what we call head pointer...

While pulling changes from remote server , first origin/master moves the pointer but ,master(local) doesnt move because we need to be

a fast forward merge to move head pointer(master-local)

origin/master is just a branch that tries to stay in sync with remote server.

## Add Remote Repositories

Remote repositories are versions of your projects that are stored on the internet or elsewhere. Adding a remote repository is a way of telling Git where your code is stored.

We can do this using the URL of the repository. This could be the URL of your repository, another user's fork, or even a completely different server.

When you clone a repository, Git implicitly adds that repository as the origin remote for you. To add a new Git repository, you use this command:

```
git remote add <shortname> <url>
```

where shortname is a unique remote name and url is the url of the repository you want to add. For example, if I want to add a repository with the shortname upstream, I can do this:

```
git remote add upstream https://github.com/GoudSagar/my\_project.git
```

Remember that your shortname can be anything, it just has to be unique, that is different from what the names of the remote repositories you already have. It should also be something you can easily remember for your sanity.

To view the list of remote URLs you have added, run the following command:

```
git remote -v
```

You'll see a list of the remote names and the URLs you have added.

But what if you want to change these remote URLs? Let's move to the next Git command.

## Change remote repositories

There are several reasons why you may want to change a remote URL. For example, I recently had to move from using https URLs to SSH URLs for a project I worked on.

To do this, you use the following command:

```
git remote set-url <an-existing-remote-name> <url>
```

For this command to work, the remote name has to be an existing remote name. That means it won't work if you've not added that remote name before.

Using the example above, if I want to change the remote URL, I'll do this:

```
git remote set-url upstream https://github.com/GoudSagar/my\_project.git
```

Remember to run `git remote -v` to verify that your change worked.  
Enough about remote repositories. Let's move on to something different.

## PULL REQUESTS:

Pull requests are used to discuss some piece of code with the other collaborators before finally merging the code. They can be useful if you are contributing to an open source project or have some people working remotely. You can specify which branch you'd like to merge your changes into when you create your pull request.

To create a pull request, first, you need to create a new branch from the branch you wish to merge into. Next, you need to commit some changes and push this branch to Github.

- **How to make a pull request:** After pushing your code to your local repository you have to make a pull request to merge your code to the real repository. To do that just go to your local repo and click on the pull request.

Follow below Procedure for pull Request:

```
MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git remote -v
origin  https://github.com/sagargoud009/Hello_World12.git (fetch)
origin  https://github.com/sagargoud009/Hello_World12.git (push)
upstream https://github.com/GoudSagar/Hello_World12.git (fetch)
upstream https://github.com/GoudSagar/Hello_World12.git (push)

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$
```

```
MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git pull upstream master
From https://github.com/GoudSagar/Hello_World12
* branch      master      -> FETCH_HEAD
Already up to date.

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
```

```
MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ ls -lrt
total 4
-rw-r--r-- 1 MyHome 197608 15 Apr 12 17:42 README.md
-rw-r--r-- 1 MyHome 197608 34 Apr 12 17:42 feedback.html
-rw-r--r-- 1 MyHome 197608 13 Apr 12 17:42 file1
-rw-r--r-- 1 MyHome 197608 23 Apr 12 17:42 form
-rw-r--r-- 1 MyHome 197608 0 Apr 12 17:43 f1
-rw-r--r-- 1 MyHome 197608 0 Apr 12 17:43 f2
-rw-r--r-- 1 MyHome 197608 0 Apr 12 17:43 f3

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ touch x1 x2 x3
```

```

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git add .

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git commit -m "Adding x1 x2 x3"
[master 54ef87b] Adding x1 x2 x3
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 x1
create mode 100644 x2
create mode 100644 x3

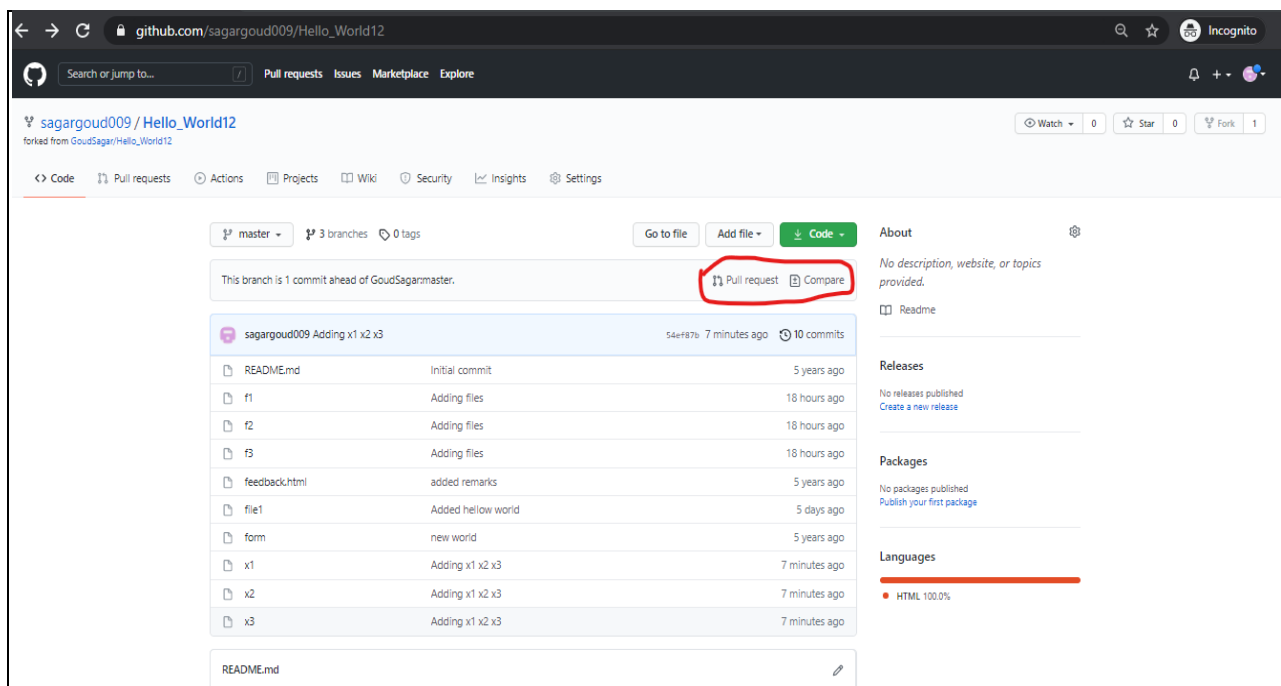
MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$ git push origin master
Logon failed, use ctrl+c to cancel basic credential prompt.
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 269 bytes | 269.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/sagargoud009/Hello_World12.git
9953eaa..54ef87b master -> master

MyHome@MyHome-PC MINGW64 ~/Documents/DEVOPS BATCH/GIT_WORKOUT/Hello_World12 (master)
$

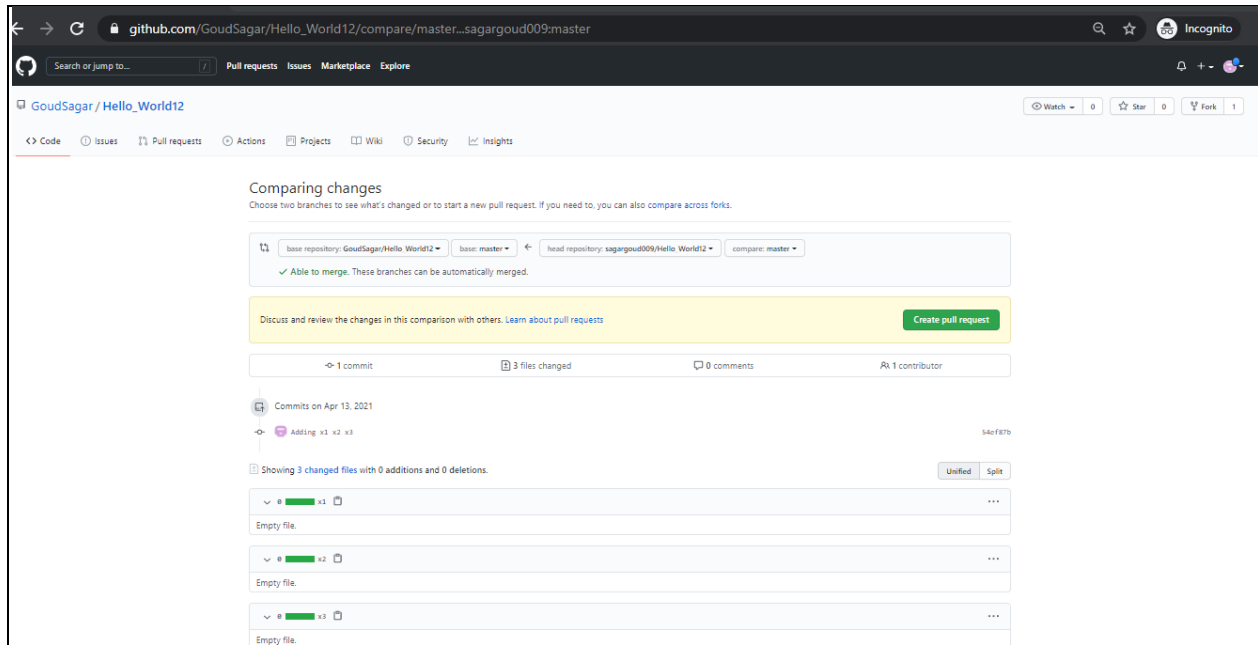
```

Go to Github Remote Repo and Create pull request again Remote Master Branch:

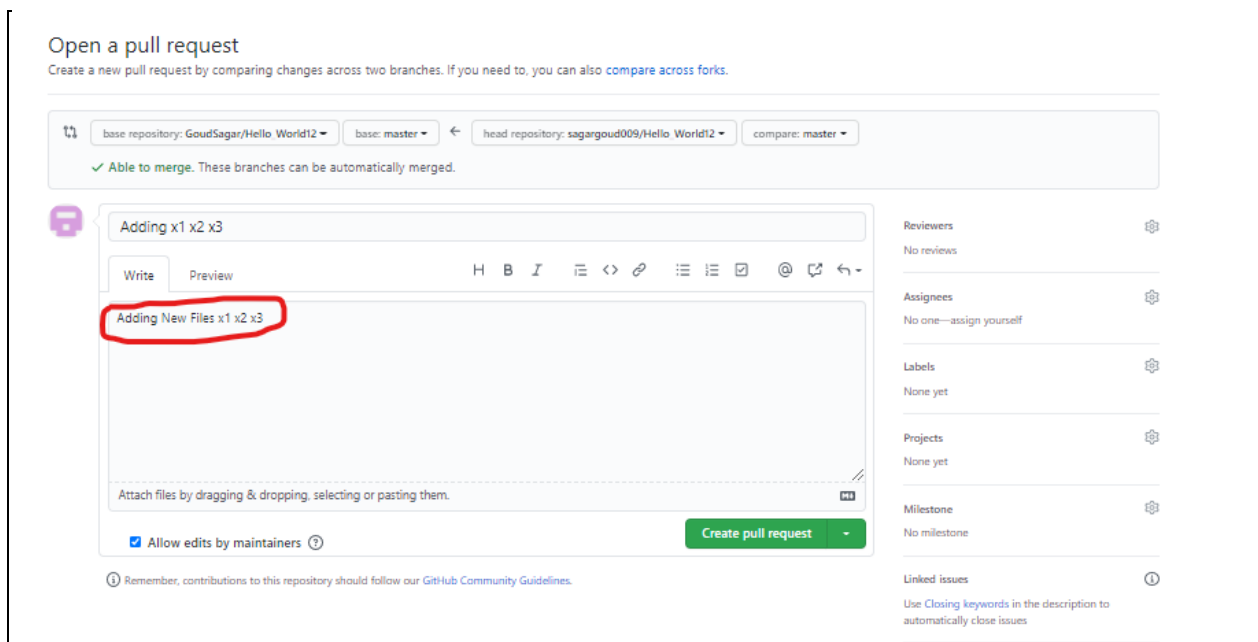
1. Click on pull request button.



2. Compare both the repo branches and click on Create pull Request

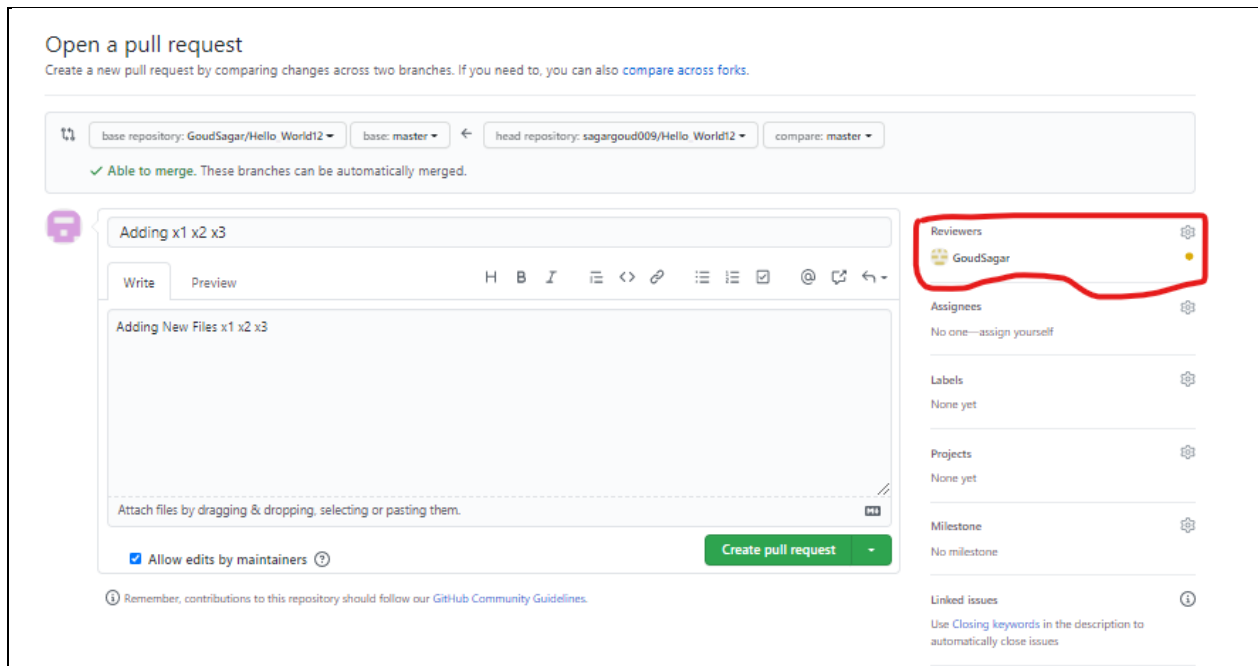


### 3. Add Message or Jira ticket details in Comment section.

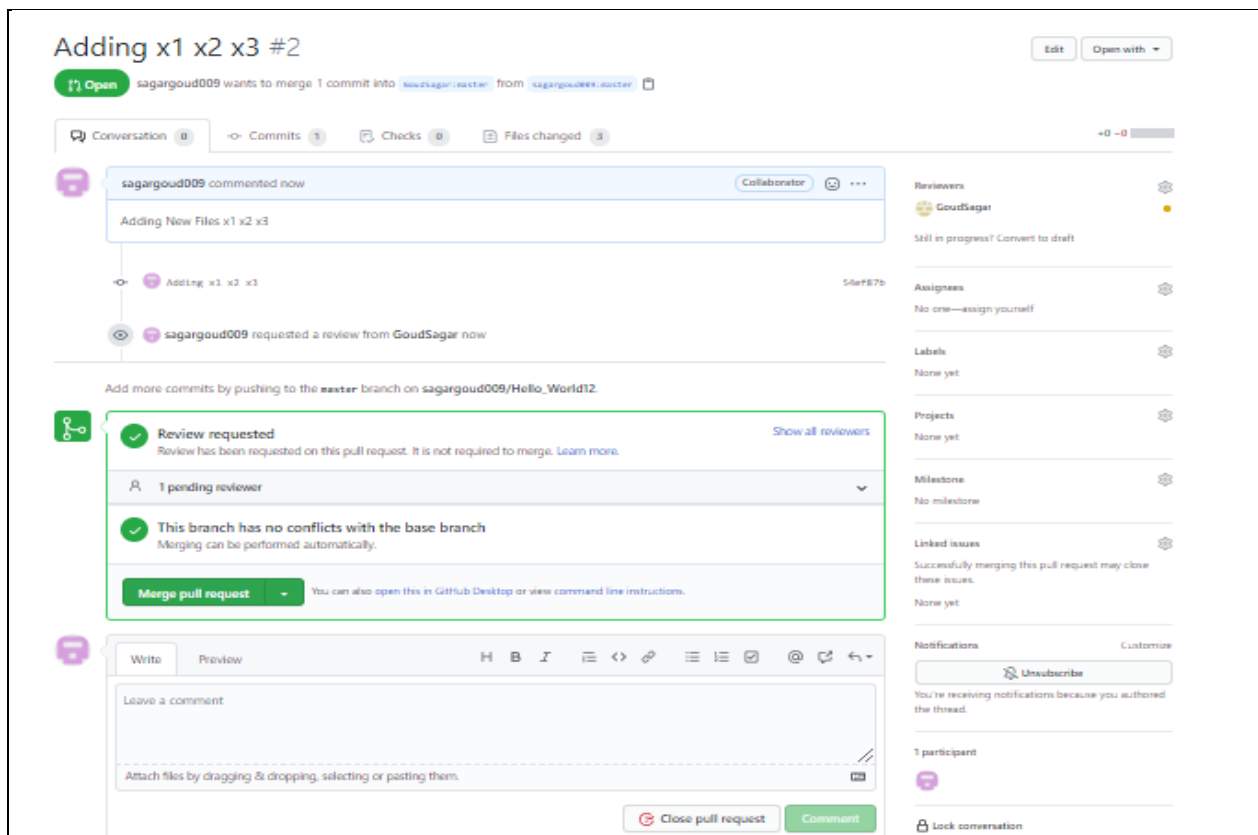


### 4. Add Reviewer to review your code and click on create pull request.

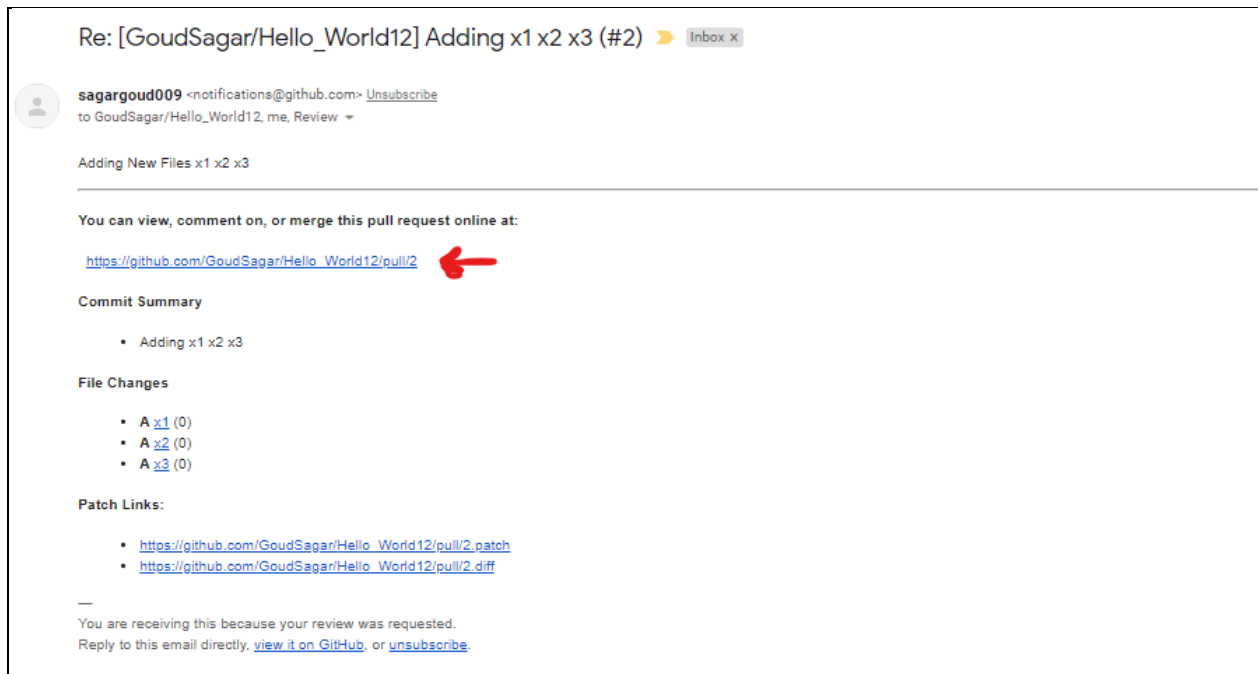




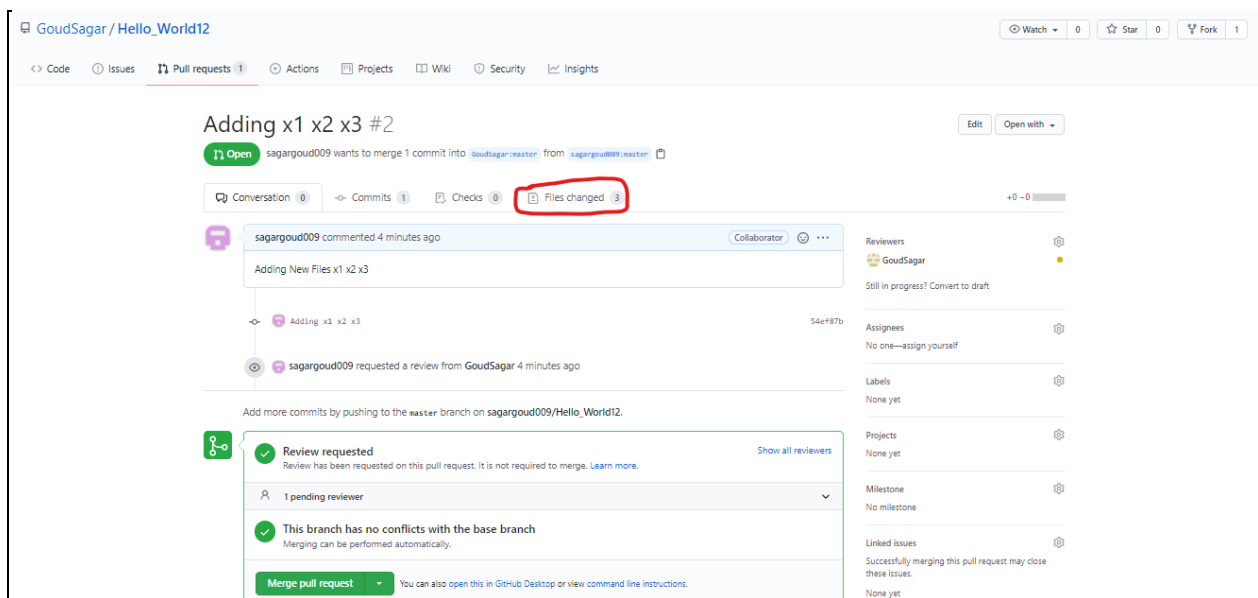
5. Finally Pull Request is raised. An email notification should triggered for Reviewer to Notify about PR



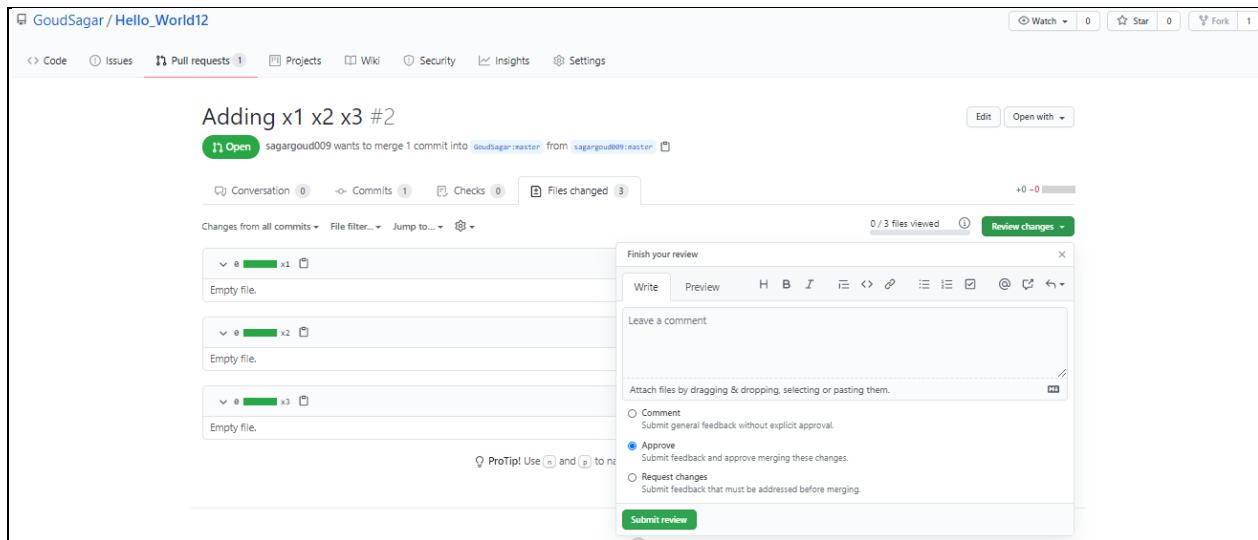
6. Email looks like below and click on PR .



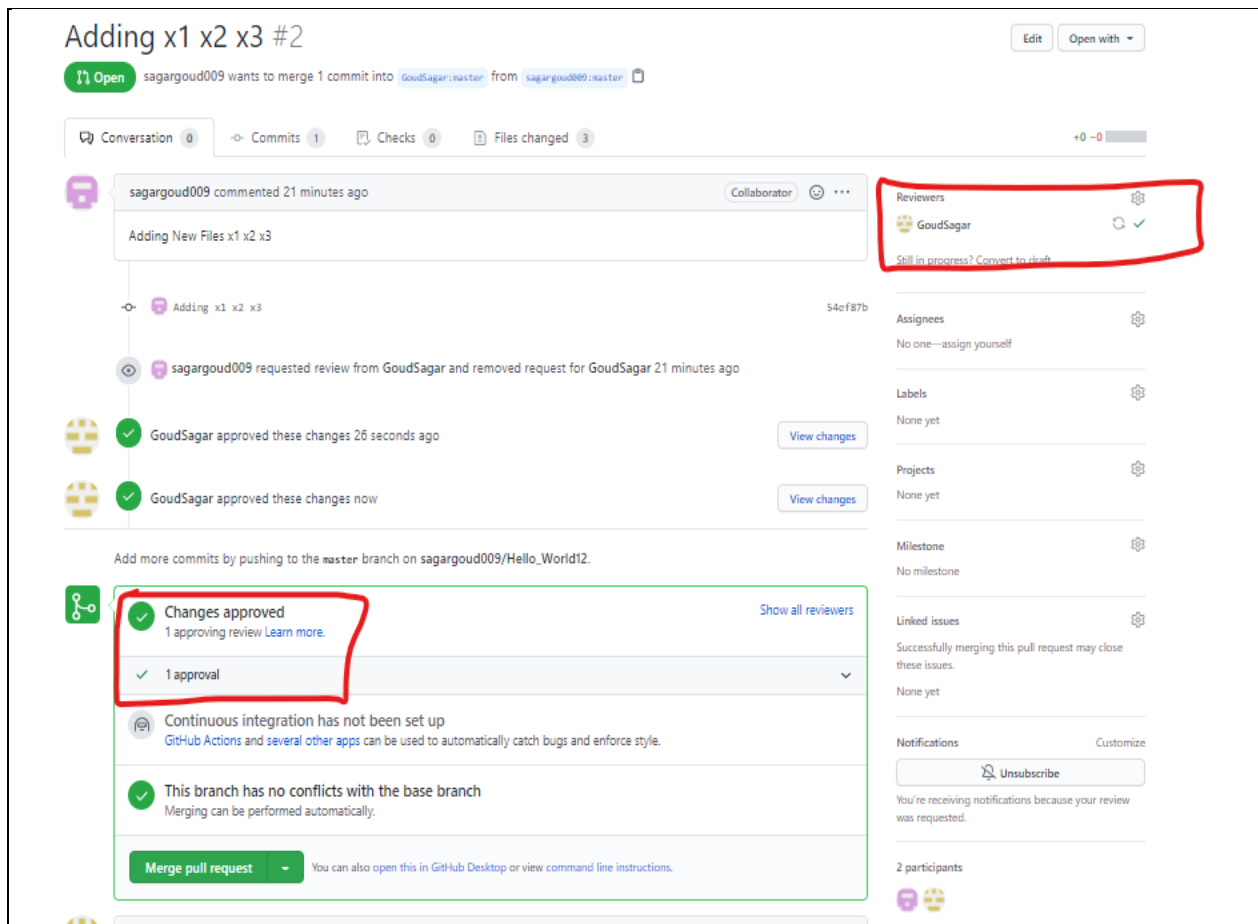
7. GitHub page opens and click on file changes tab:

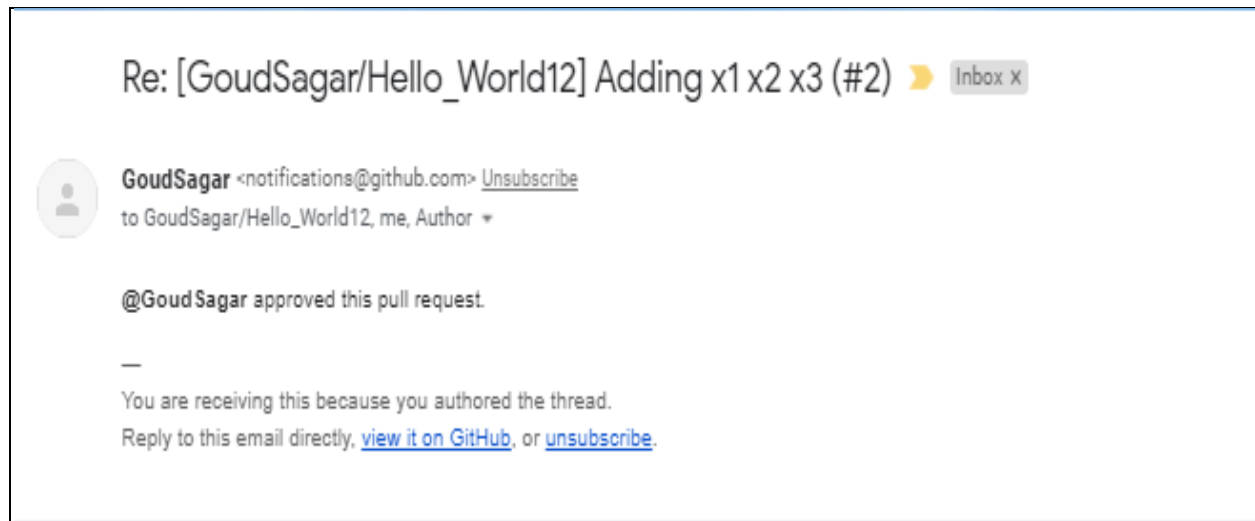


8. Click on Review Changes, Add comment/Approve/Reject and Submit Review.

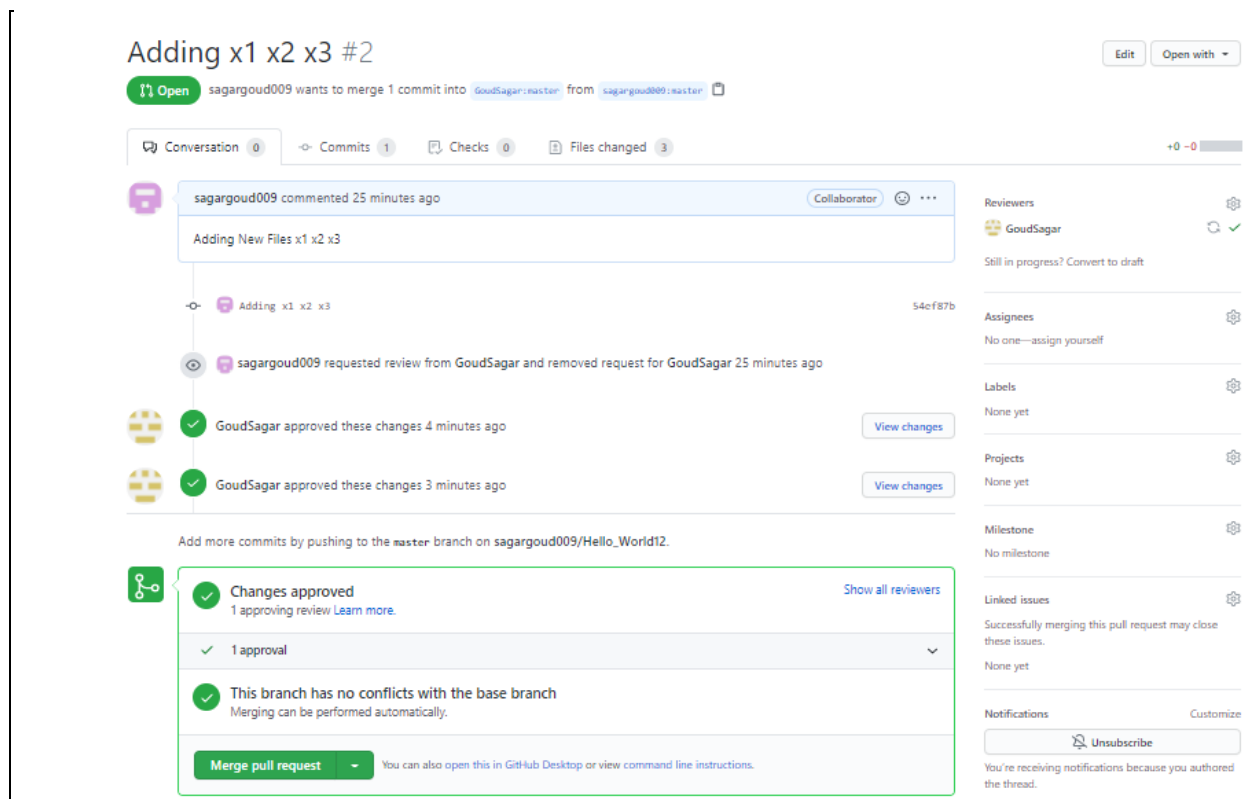


9. Github page looks like below and email notification will review to user who raises PR.





11. Click on Merge pull request for merging into Remote Master Branch.



12. Click Confirm Merge

## Adding x1 x2 x3 #2

Open

sagargoud009 wants to merge 1 commit into `GoudSagar:master` from `sagargoud009:master`

Conversation 0

Commits 1

Checks 0

Files changed 3

+0 -0

sagargoud009 commented 26 minutes ago

Collaborator

...

Adding New Files x1 x2 x3

Adding x1 x2 x3

S4ef87b

sagargoud009 requested review from GoudSagar and removed request for GoudSagar 26 minutes ago

GoudSagar approved these changes 5 minutes ago

View changes

GoudSagar approved these changes 4 minutes ago

View changes

Add more commits by pushing to the `master` branch on `sagargoud009/Hello_World12`.

Merge pull request #2 from `sagargoud009/master`

Adding x1 x2 x3

Confirm merge

Cancel

Write

Preview

H B I

< >

Reviewers

GoudSagar

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Linked issues

Successfully merging this pull request may close these issues.

None yet

Notifications

Unsubscribe

You're receiving notifications because you authored the thread.

13.Pull Request is Merged.

53

## Adding x1 x2 x3 #2

**Merged** sagargoud009 merged 1 commit into GoudSagar:master from sagargoud009:master now

Conversation 0 Commits 1 Checks 0 Files changed 3 +0 -0

sagargoud009 commented 27 minutes ago

Adding New Files x1 x2 x3

Adding x1 x2 x3 54ef87b

sagargoud009 requested review from GoudSagar and removed request for GoudSagar 27 minutes ago

GoudSagar approved these changes 6 minutes ago [View changes](#)

GoudSagar approved these changes 5 minutes ago [View changes](#)

sagargoud009 merged commit 0988377 into GoudSagar:master now [Revert](#)

**Pull request closed**  
If you wish, you can delete this fork of GoudSagar/Hello\_World12 in the [settings](#).

Write Preview [H](#) [B](#) [I](#) [≡](#) [<>](#) [🔗](#) [⋮](#) [⋮](#) [☑](#) [@](#) [🗨](#) [↩](#)

Leave a comment

**Reviewers**  
GoudSagar ✓

**Assignees**  
No one—assign yourself

**Labels**  
None yet

**Projects**  
None yet

**Milestone**  
No milestone

**Linked issues**  
Successfully merging this pull request may close these issues.  
None yet

**Notifications** [Customize](#)  
[Unsubscribe](#)  
You're receiving notifications because you modified the open/close state.

#### 14. Check code and Commits on Remote Master Branch.

GoudSagar / Hello\_World12 [Watch](#)

[Code](#) [Issues](#) [Pull requests](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

master 3 branches 0 tags [Go to file](#) [Add file](#) [Code](#)

sagargoud009 Merge pull request #2 from sagargoud009/master 0988377 1 minute ago 11 commits

README.md	Initial commit	5 years ago
f1	Adding files	18 hours ago
f2	Adding files	18 hours ago
f3	Adding files	18 hours ago
feedback.html	added remarks	5 years ago
file1	Added hellow world	5 days ago
form	new world	5 years ago
x1	Adding x1 x2 x3	44 minutes ago
x2	Adding x1 x2 x3	44 minutes ago
x3	Adding x1 x2 x3	44 minutes ago

README.md

Hello\_World12

**About**  
No description, website, or topics provided.

**Releases**  
No releases published  
[Create a new release](#)

**Packages**  
No packages published  
[Publish your first package](#)

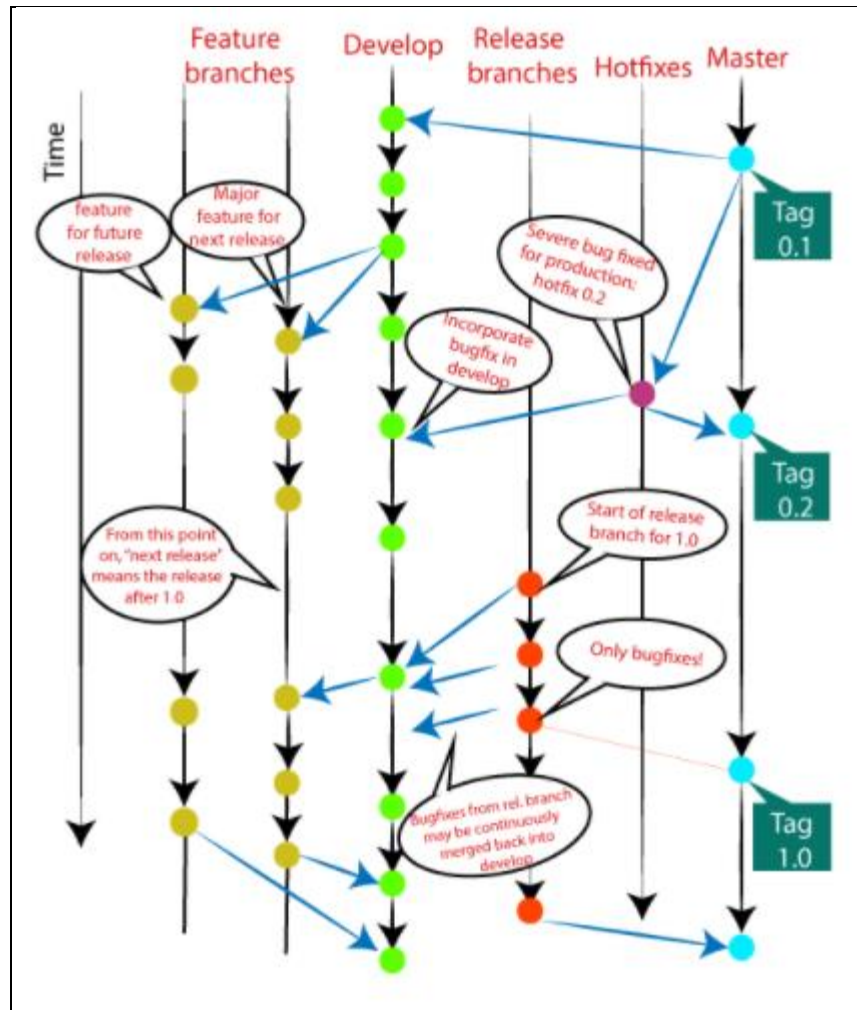
**Contributors** 2  
GoudSagar  
sagargoud009

**Languages**  
HTML 100.0%

## Git Flow / Git Branching Model

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.



There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

- **Master**
- **Develop**
- **Hotfixes**

- **Release branches**
- **Feature branches**

Every branch has its meaning and standard. Let's understand each branch and its usage.

## The Main Branches

Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

- **master**
- **develop**

## Master Branch

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

## Develop Branch

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "**integration branch**."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

## Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

The different types of supportive branches, we may use are as follows:

- **Feature branches**
- **Release branches**



- **Hotfix branches**

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

## **Feature Branches**

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.

## **Release Branches**

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

## **Hotfix Branches**

Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.

## **Git Merge vs. Rebase**

It is a most common puzzling question for the git user's that when to use merge command and when to use rebase. Both commands are similar, and both are used to merge the commits made by the different branches of a repository.

Rebasing is not recommended in a shared branch because the rebasing process will create inconsistent repositories. For individuals, rebasing can be more useful than merging. If you want to see the complete history, you should use the merge. Merge tracks the entire history of commits, while rebase rewrites a new one.

Git rebase commands said as an alternative of git merge. However, they have some key differences:

Git Merge	Git Rebase
Merging creates a final commit at merging.	Git rebase does not create any commit at rebasing.
It merges all commits as a single commit.	It creates a linear track of commits.
It creates a graphical history that might be a bit complex to understand.	It creates a linear history that can be easily understood.
It is safe to merge two branches.	Git "rebase" deals with the severe operation.
Merging can be performed on both public and private branches.	It is the wrong choice to use rebasing on public branches.
Merging integrates the content of the feature branch with the master branch. So, the master branch is changed, and feature branch history remains consistence.	Rebasing of the master branch may affect the feature branch.
Merging preserves history.	Rebasing rewrites history.
Git merge presents all conflicts at once.	Git rebase presents conflicts one by one.