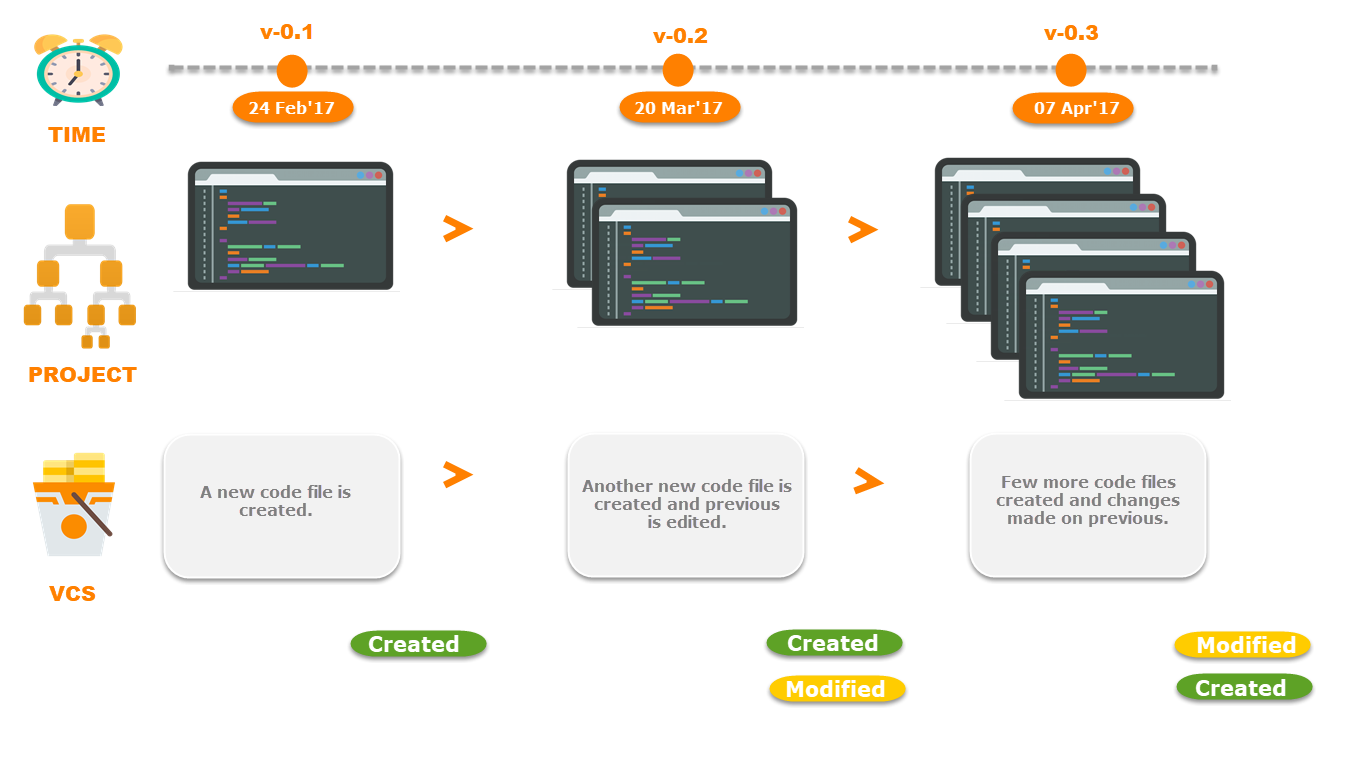
**Git Full Documentation**

**What is Version Control System?**

You can think of a ***Version Control System*** (aka: ***VCS***) as a kind of ***database***. It lets you save a snapshot of the complete project at any point of time. Every change made to the project files is tracked, along with who made the change, why they made it, and references to problems fixed, or enhancements introduced, by the change. Later when it is required to take a look at an older snapshot/version, VCS shows how exactly it differed from the previous one.

***Version Control Systems (VCS) are tools that help teams manage and track changes in code over time****.*



With reference to the image above, on 24 Feb, 17 your project has a new file added to it. This file could be a source-code file, a properties file, an image or any other type of file. Once your project is tracked by VCS, any addition, edit or deletion of files from your project will be automatically detected and recorded by it. In short, every time a change is made to the project, VCS creates and stores a snapshot in form of versions.

**There are two types of VCS:**

* Centralized Version Control System (CVCS)
* Distributed Version Control System (DVCS)

**Centralized VCS**

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.

Please refer to the diagram below to get a better idea of CVCS:



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.

**Distributed VCS**

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.

## Working with local repositories

### **gitconfig**

With Git, there are many configurations and settings possible. gitconfig is how to assign these settings. Two important settings are user user.name and user.email. These values set what email address and name commits will be from on a local computer. With gitconfig, a --global flag is used to write the settings to all repositories on a computer. Without a --global flag settings will only apply to the current repository that you are currently in.

There are many other variables available to edit in gitconfig. From editing color outputs to changing the behavior of git status. Learn about gitconfig settings in the official [Git documentation](https://git-scm.com/docs/git-config).

Usage:

$ gitconfig<setting><command>

In Practice:

# Runninggitconfig globally

$ gitconfig --global user.email "my@emailaddress.com"

$ gitconfig --global user.name "Brian Kerr"

# Runninggitconfig on the current repository settings

$ gitconfiguser.email "my@emailaddress.com"

$ gitconfig user.name "Brian Kerr"

### **gitinit**

This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running gitinit, adding and committing files/directories is possible.

Usage:

# change directory to codebase

$ cd /file/path/to/code

# make directory a git repository

$ gitinit

In Practice:

# change directory to codebase

$ cd /Users/computer-name/Documents/website

# make directory a git repository

$ gitinit

Initialized empty Git repository in /Users/computer-name/Documents/website/.git/

### **git add**

Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files.

Usage:

$ git add <file or directory name>

In Practice:

# To add all files not staged:

$ git add .

# To stage a specific file:

$ git add index.html

# To stage an entire directory:

$ git add css

### **git commit**

Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.

It’s best practice to include a message with each commit explaining the changes made in a commit. Adding a commit message helps to find a particular change or understanding the changes.

Usage:

# Adding a commit with message

$ git commit -m "Commit message in quotes"

In Practice:

$ git commit -m "My first commit message"

[SecretTesting 0254c3d] My first commit message

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 homepage/index.html

### **git status**

This command returns the current state of the repository.

git status will return the current working branch. If a file is in the staging area, but not committed, it shows with git status. Or, if there are no changes it’ll return nothing to commit, working directory clean.

Usage:

$ git status

In Practice:

# Message when files have not been staged (git add)

$ git status

On branch SecretTesting

Untracked files:

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

homepage/index.html

# Message when files have been not been committed (git commit)

$ git status

On branch SecretTesting

Your branch is up-to-date with 'origin/SecretTesting'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: homepage/index.html

# Message when all files have been staged and committed

$ git status

On branch SecretTesting

nothing to commit, working directory clean

### **git branch**

To determine what branch the local repository is on, add a new branch, or delete a branch.

Usage:

# Create a new branch

$ git branch <branch\_name>

# List all remote or local branches

$ git branch -a

# Delete a branch

$ git branch -d <branch\_name>

In Practice:

# Create a new branch

$ git branch new\_feature

# List branches

$ git branch -a

\* SecretTesting

new\_feature

remotes/origin/stable

remotes/origin/staging

remotes/origin/master -> origin/SecretTesting

# Delete a branch

$ git branch -d new\_feature

Deleted branch new\_feature (was 0254c3d).

### **git checkout**

To start working in a different branch, use git checkout to switch branches.

Usage:

# Checkout an existing branch

$ git checkout <branch\_name>

# Checkout and create a new branch with that name

$ git checkout -b <new\_branch>

In Practice:

# Switching to branch 'new\_feature'

$ git checkout new\_feature

Switched to branch 'new\_feature'

# Creating and switching to branch 'staging'

$ git checkout -b staging

Switched to a new branch 'staging'

### **git merge**

Integrate branches together. git merge combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch.

Usage:

# Merge changes into current branch

$ git merge <branch\_name>

In Practice:

# Merge changes into current branch

$ git merge new\_feature

Updating 0254c3d..4c0f37c

Fast-forward

homepage/index.html | 297 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

1 file changed, 297 insertions(+)

create mode 100644 homepage/index.html

**Git bare repository:**

The standard way of initializing a new Git repository is to run gitinit. The directory in which you do this will be become the [Working Tree](https://mijingo.com/blog/what-is-the-working-tree-in-git) for the repository.

As part of the initialization process, Git creates a .git directory (which his hidden by default because of the . in the name) that contains the repository itself. This is brains of the repository; it's where Git tracks your changes, stores commit objects, refs, etc. You probably only rarely interact with that hidden directory.

Okay, so all of this is to lay the groundwork for understanding a bare Git repository. What the heck is it?

A bare Git repository is a repository that is created without a Working Tree. Go ahead and create one to see.

gitinit --bare .

Run ls on that directory and you won't see a Working Tree but just the contents of what is typically in the .git directory.

**Why this setup?**

A bare Git repository is typically used as a [Remote Repository](https://mijingo.com/blog/what-is-a-git-remote-repository) that is sharing a repository among several different people. You don't do work right inside the remote repository so there's no Working Tree (the files in your project that you edit), just bare repository data.

**Why use one or the other?**

A working repository created with gitinit is for… **working**. It is where you will actually edit, add and delete files and git commit to save your changes. If you are starting a project in a folder on your dev machine where you will add, edit and delete files of your project, use “gitinit”. Note: if you git clone a repository you will be given a **working** repository with the .git folder and copies of the working files for editing.

A bare repository created with gitinit --bare is for… **sharing**. If you are collaborating with a team of developers, and need a place to share changes to a repo, then you will want to create a bare repository in centralized place where all users can push their changes (often the easy choice is [**github.com**](http://github.com/)). Because git is a distributed version control system, no one will directly edit files in the shared centralized repository. Instead developers will clone the shared bare repo, make changes locally in their working copies of the repo, then push back to the shared bare repo to make their changes available to other users.

Because no one ever makes edits directly to files in the shared bare repo, a working tree is not needed. In fact the working tree would just get in way and cause conflicts as users push code to the repository. This is why bare repositories exist and have no working tree.

## Advanced Git Commands

### **git stash**

To save changes made when they’re not in a state to commit them to a repository. This will store the work and give a clean working directory. For instance, when working on a new feature that’s not complete, but an urgent bug needs attention.

Usage:

# Store current work with untracked files

$ git stash -u

# Bring stashed work back to the working directory

$ git stash pop

In Practice:

# Store current work

$ git stash -u

Saved working directory and index state WIP on SecretTesting: 4c0f37c Adding new file to branch

HEAD is now at 4c0f37c Adding new file to branch

# Bring stashed work back to the working directory

$ git stash pop

On branch SecretTesting

Your branch and 'origin/SecretTesting' have diverged,

and have 1 and 1 different commit each, respectively.

(use "git pull" to merge the remote branch into yours)

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: index.html

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (3561897724c1f448ae001edf3ef57415778755ec)

### **git log**

To show the chronological commit history for a repository. This helps give context and history for a repository. git log is available immediately on a recently cloned repository to see history.

Usage:

# Show entire git log

$ git log

# Show git log with date pameters

$ git log --<after/before/since/until>=<date>

# Show git log based on commit author

$ git log --<author>="Author Name"

In Practice:

# Show entire git log

$ git log

commit 4c0f37c711623d20fc60b9cbcf393d515945952f

Author: Brian Kerr <my@emailaddress.com>

Date: Tue Oct 25 17:46:11 2016 -0500

Updating the wording of the homepage footer

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67

Author: Ashley Harpp<my@emailaddress.com>

Date: Wed Oct 19 16:27:27 2016 -0500

My first commit message

# Show git log with date pameters

$ git log --before="Oct 20"

commit 0254c3da3add4ebe9d7e1f2e76f015a209e1ef67

Author: Ashley Harpp<my@emailaddress.com>

Date: Wed Oct 19 16:27:27 2016 -0500

My first commit message

# Show git log based on commit author

$ git log --author="Brian Kerr"

commit 4c0f37c711623d20fc60b9cbcf393d515945952f

Author: Brian Kerr <my@emailaddress.com>

Date: Tue Oct 25 17:46:11 2016 -0500

Updating the wording of the homepage footer

### **gitrm**

Remove files or directories from the working index (staging area). With gitrm, there are two options to keep in mind: force and cached. Running the command with force deletes the file. The cached command removes the file from the working index. When removing an entire directory, a recursive command is necessary.

Usage:

# To remove a file from the working index (cached):

$ gitrm --cached <file name>

# To delete a file (force):

$ gitrm -f <file name>

# To remove an entire directory from the working index (cached):

$ gitrm -r --cached <directory name>

# To delete an entire directory (force):

$ gitrm -r -f <file name>

In Practice:

# To remove a file from the working index:

$ gitrm --cached css/style.css

rm 'css/style.css'

# To delete a file (force):

$ gitrm -f css/style.css

rm 'css/style.css'

# To remove an entire directory from the working index (cached):

$ gitrm -r --cached css/

rm 'css/style.css'

rm 'css/style.min.css'

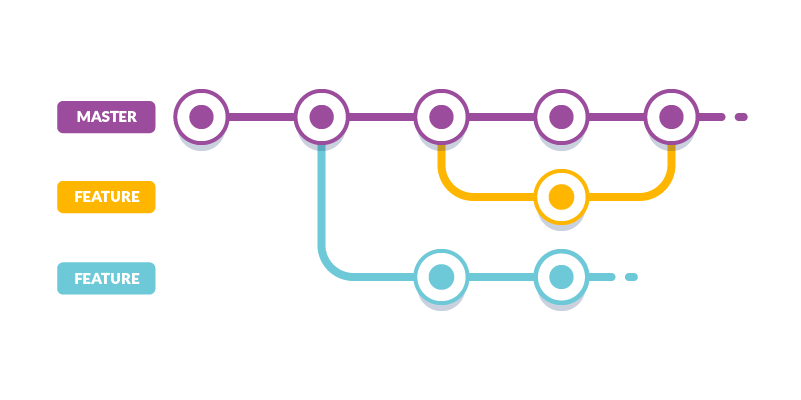
# To delete an entire directory (force):

$ gitrm -r -f css/

rm 'css/style.css'

rm 'css/style.min.css'

# **How to adopt a Git branching strategy:**



git branching strategy

Distributed version control systems like Git give individuals wide flexibility in how they use version control to share and manage code. Your team should find a balance between this flexibility and the need to collaborate so that you share code in a consistent manner.

#### **Why bother with branching?**

Branching allows a team of developers to collaborate inside of one central code base. Team members publish, share, review and iterate on code changes through Git branches shared with others.

Adopt a branching strategy for your team. You collaborate better and spend less time managing version control, and your team spends more time developing.

***Keep it simple***

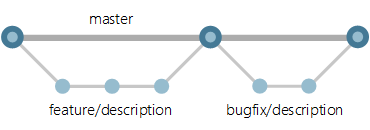
Keep your branch strategy simple by building from these three concepts:

* Use feature branches for all new features and bug fixes
* Merge feature branches into the master branch using pull requests
* Keep master branch up-to-date

A strategy that extends these concepts and avoids contradictions will result in a version control workflow for your team that is consistent and easy to follow.

#### **Feature branch strategy**

Develop your features and fix bugs in feature branches based off your master branch (also known as topic branches). Feature branches isolate work in progress from the completed work in the master branch. Git branches are inexpensive to create and maintain. Even small fixes and changes should have their own feature branch.



Feature branching

Creating feature branches for all your changes makes reviewing history very simple. You can have a clean commit history which can be helpful to debug (or) revert when needed.

#### **Branch naming convention**

Use a consistent naming convention for your feature branches to identify the work done in the branch.

A Few suggestions for naming feature branches:   
\* feature/feature-name   
\* feature/feature-area/feature-name   
\* bugfix/description   
\* hotfix/description

feature : New feature   
bugfix : Changes linked to a known issue   
hotfix : Quick fixes to the codebase

#### **Review and Merge**

The review of a pull request is critical for improving code quality. Only merge branches through pull requests that pass your review process. Avoid merging branches to the master branch without a pull request.

Reviews in pull requests take time to complete. Your team should agree on what's expected from pull request creators and reviewers. Distribute reviewer responsibility across your team and spread out knowledge of your codebase.

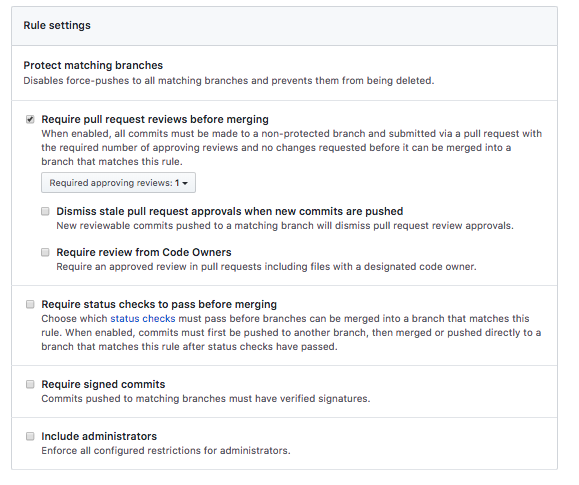
**Research says that conducting peer review increases the number of distinct files a developer knows about by 66% to 150% depending on the project.**

A few suggestions for successful pull requests:

* Two reviewers is an optimal number
* Take care not to assign the same reviewer(s) to a large number of pull requests. Pull requests work better when a reviewer’s responsibilities are shared across the team.
* Provide enough detail in the description about the change
* Include a build (or) linked version of your changes running in a staged environment with your changes so others can easily test

#### **Branch Policy**

Branch policies help teams protect their important branches of development. Policies enforce your team's code quality and standards. If you are using **Github,** here is how the branch policies look like:



Github repository branch policies

**Manage releases**Once you have a stable master branch with up-to-date code, managing releases can be done either by maintaining **release branches** or **tags.**

**Release Branches**Create a release branch from the master as you get close to your release or other milestones, such as the end of the sprint. Give this branch a clear name associating it with a release, for example release/2.0 .

**Tags**Tags are useful for marking points in your history as important. Tags can be used instead of release branches and push the fixes into master as hotfixes through pull requests.

***Conclusion***

A git branching strategy is important to have a smooth development process. Defining how you branch, review, and release can help your team move fast.

I hope you’ve gotten a basic understanding about Git branching strategy. I’m looking forward to writing on **Git workflow** and concepts of **Git**. Comment on a topic that you want me to write about next. Cheers!

### **Useful commands**

#### 🔍 Status

Check the status of working directory and staging area:

git status

Show changes between HEAD and working directory:

git diff

Show the list of commits in one line format:

git log --oneline

Show commits that make add or remove a certain string:

git log -S 'LoginViewController'

Search commits that contain a log message:

git log — all — grep=’day of week’

#### 🔍 Tag

List all tags:

git tag

Tag a commit:

git tag -a 1.4 -m "my version 1.4"

Delete remote tags:

git push --delete origin tagname

git push origin :tagname

Push tag to remote:

git push origin tagname

Rename tag:

git tag new old  
git tag -d old  
git push origin :refs/tags/old  
git push --tags

Move tag from one commit to another commit:

git push origin :refs/tags/<tagname>  
git tag -fa tagname  
git push origin master --tags

#### 🔍 Remote

List all remote:

git remote

Rename remote:

git remote rename old new

Remove stale remote tracking branches:

git remote prune origin

#### 🔍 Branch

List all branches:

git branch

Create the branch on your local machine and switch in this branch:

git checkout -b branch\_name

Create branch from commit:

git branch branch\_name sha1\_of\_commit

Push the branch to remote:

git push origin branch\_name

Rename other branch:

git branch -m old new

Rename current branch:

git branch -m new

Rename remote branch:

git branch -m old new # Rename branch locally   
git push origin :old # Delete the old branch   
git push --set-upstream origin new # Push the new branch, set local branch to track the new remote

Delete a branch:

git branch -D the\_local\_branch

git push origin :the\_remote\_branch

**git branch -d <branchname> ---🡪 (**Delete the feature branch:**)**

Delete all local branches but master

git branch | grep -v "master" | xargsgit branch -D

#### 🔍 Commit

Undo last commit:

git reset --hard HEAD~1

Squash last n commits into one commit:

git rebase -i HEAD~5

git reset --soft HEAD~5  
git add .  
git commit -m "Update"  
git push -f origin master

Move last commits into new branch:

git branch newbranch  
git reset --hard HEAD~3 # Go back 3 commits. You \*will\* lose uncommitted work.\*1  
git checkout newbranch

#### 🔍 Cherry Pick

Add some commits to the top of the current branch:

git cherry-pick hash\_commit\_Ahash\_commit\_B

#### 🔍Reflog

Show reflog:

gitreflog

Get commit:

git reset --hard 0254ea7

git cherry-pick 12944d8

#### 🔍 Revert

Revert the previous commit:

git revert HEAD  
git commit

Revert the changes from previous 3 commits without making commit:

git revert --no-commit HEAD~3..

#### 🔍 Amend

Amend previous commit:

git commit --amend

git commit --amend --no-edit

git commit --amend -m "New commit message"

[Changing git commit message after push](http://stackoverflow.com/questions/8981194/changing-git-commit-message-after-push-given-that-no-one-pulled-from-remote):

git commit --amend -m "New commit message"  
git push --force <repository><branch>

#### 🔍 Checkout

Checkout a tag:

git checkout tagname

git checkout -b newbranchnametagname

Checkout a branch:

git checkout destination\_branch

Use -m if there is merge conflict:

git checkout -m master // from feature branch to master

Checkout a commit:

git checkout commit\_hash

git checkout -b newbranchname HEAD~4

git checkout -b newbranchnamecommit\_hash

git checkout commit\_hash file

Checkout a file:

git checkout c5f567 -- Relative/Path/To/File

#### 🔍 Stash

Save a change to stash:

git stash save "stash name"

git stash

List all stashes:

git stash list

Apply a stash:

git stash pop

git stash apply

git stash apply stash@{2}

#### 🔍 Rebase

Rebase the current branch onto master:

git rebase master // rebase the current branch onto master

Continue rebase:

git rebase --continue

Abort rebase:

git rebase --abort

#### 🔍 .gitignore

Un-track files that have just been declared in .gitignore:

gitrm -r --cached .  
git add .  
git commit -am "Remove ignored files"

#### 🔍 Index

Remove untracked files:

git clean

Remove file from index:

git reset file

Reset the index to match the most recent commit:

git reset

Reset the index and the working directory to match the most recent commit:

git reset --hard

#### 🔍Misc

Get their changes during git rebase:

git checkout --ours foo/bar.java  
git add foo/bar.java

Get their changes during git merge:

git pull -X theirs

git checkout --theirs path/to/the/conflicted\_file.php

git checkout --theirs .  
git add .

git checkout branchA  
git merge -X theirs branchB

Merge commits from master into feature branch:

git checkout feature1  
git merge --no-ff master

Find bug in commit history in a binary search tree style:

git bisect start

git bisect good

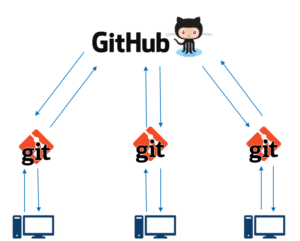
git bisect bad

## ****Introduction to GitHub****

To be very crisp about it, GitHub is a file or code sharing service to collaborate with different people.

GitHub is a highly used software which is typically used for version control. It is helpful when more than just one person is working on a project. Say for example, a software developer team wants to build a website and everyone has to update their codes simultaneously while working on the project. In this case, Github helps them to build a centralized repository where everyone can upload, edit and manage the code files.

GitHub has various advantages but many people often have a doubt as to why not use dropbox or any cloud based system? Let me take the same example forward to answer this question. Say more than two software developers are working on the same file and they want to update it simultaneously. Unfortunately, the person who save the file first will get precedence over the others. While in Github, this is not the case. Github document the changes and reflect them in an organized manner to avoid any chaos between any of the files uploaded.  
Therefore using GitHub centralized repository, it avoids all the confusion and working on the same code becomes very easy.

If you look at the image on the left, GitHub is a central repository and Git is a tool which allows you to create a local repository. Now people usually get confused between git and GitHub but its actually very different. Git is a version control tool that will allow you to perform all kinds of operations to fetch data from the central server or push data to it whereas GitHub is a core hosting platform for version control collaboration. GitHub is a company that allows you to host a central repository in a remote server.

Now let me list down the ways in which GitHub makes git simple:

* + GitHub provides you a beautiful visual interface which helps you to track or manage your version controlled projects locally.
  + Once you register on GitHub, you can connect with social network and build a strong profile.

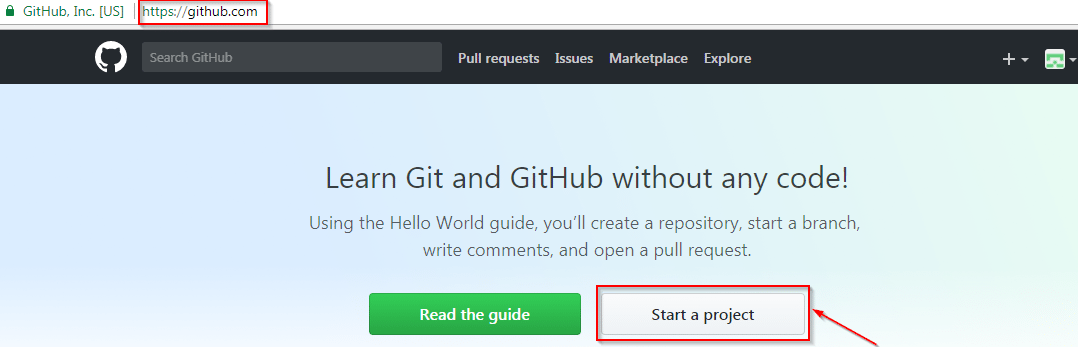
So let’s get started with GitHub.

## ****Step 2: Creating a GitHub Repository****

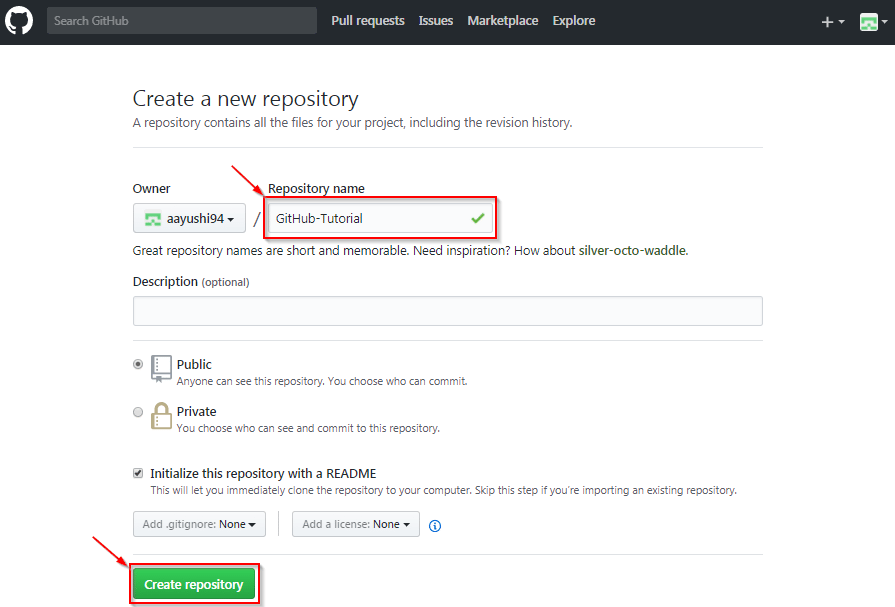
A repository is a storage space where your project lives. It can be local to a folder on your computer, or it can be a storage space on GitHub  or another online host. You can keep code files, text files, images or any kind of a file in a repository. You need a GitHub repository when you have done some changes and are ready to be uploaded. This GitHub repository acts as your remote repository. So let me make your task easy, just follow these simple steps to create a GitHub repository:

* Go to the link: <https://github.com/> . Fill the sign up form and click on “Sign up for Github”.
* Click on “Start a new project”.

Refer to the below screenshot to get a better understanding.

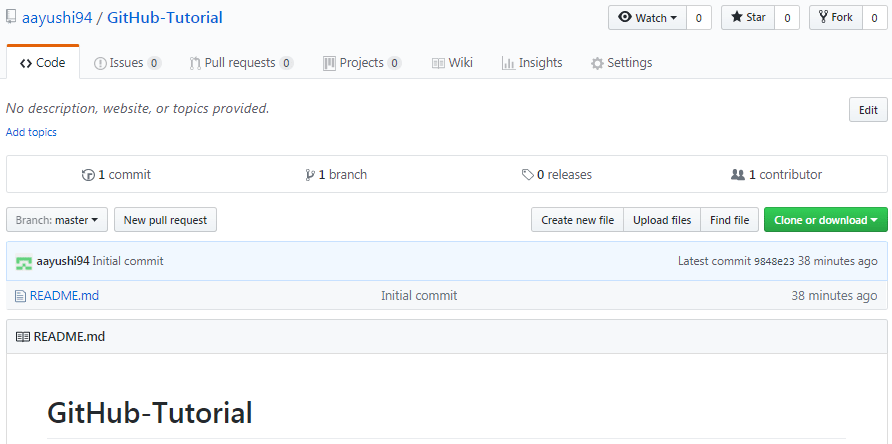


* Enter any repository name and click on “Create Repository”. You can also give a description to your repository (optional).



Now, if you noticed by default a GitHub repository is public which means that anyone can view the contents of this repository whereas in a private repository, you can choose who can view the content. Also, private repository is a paid version. Also, if you refer the above screenshot, initialize the repository with a README file. This file contains the description of the file and once you check this box, this will be the first file inside your repository.

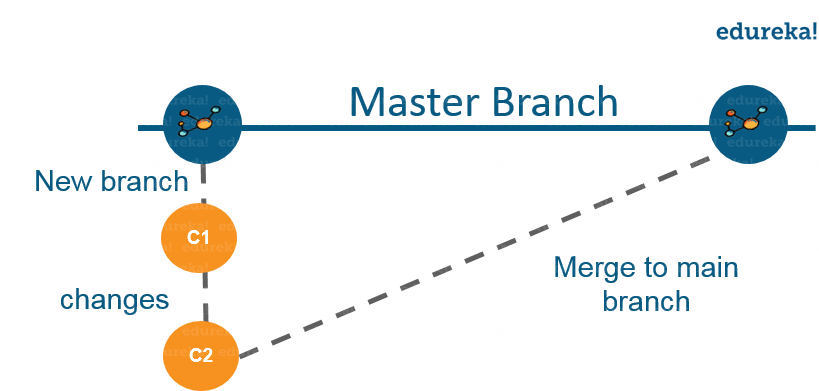
Congratulations, your repository is successfully created! It will look like the below screenshot:



So now my central repository has been sucessfully created! Once this is done, you are ready to commit, pull, push and perform all the other operations. Now let’s move forward and understand branching in GitHub.

**[Learn DevOps From Experts](https://www.edureka.co/devops" \t "_blank)**

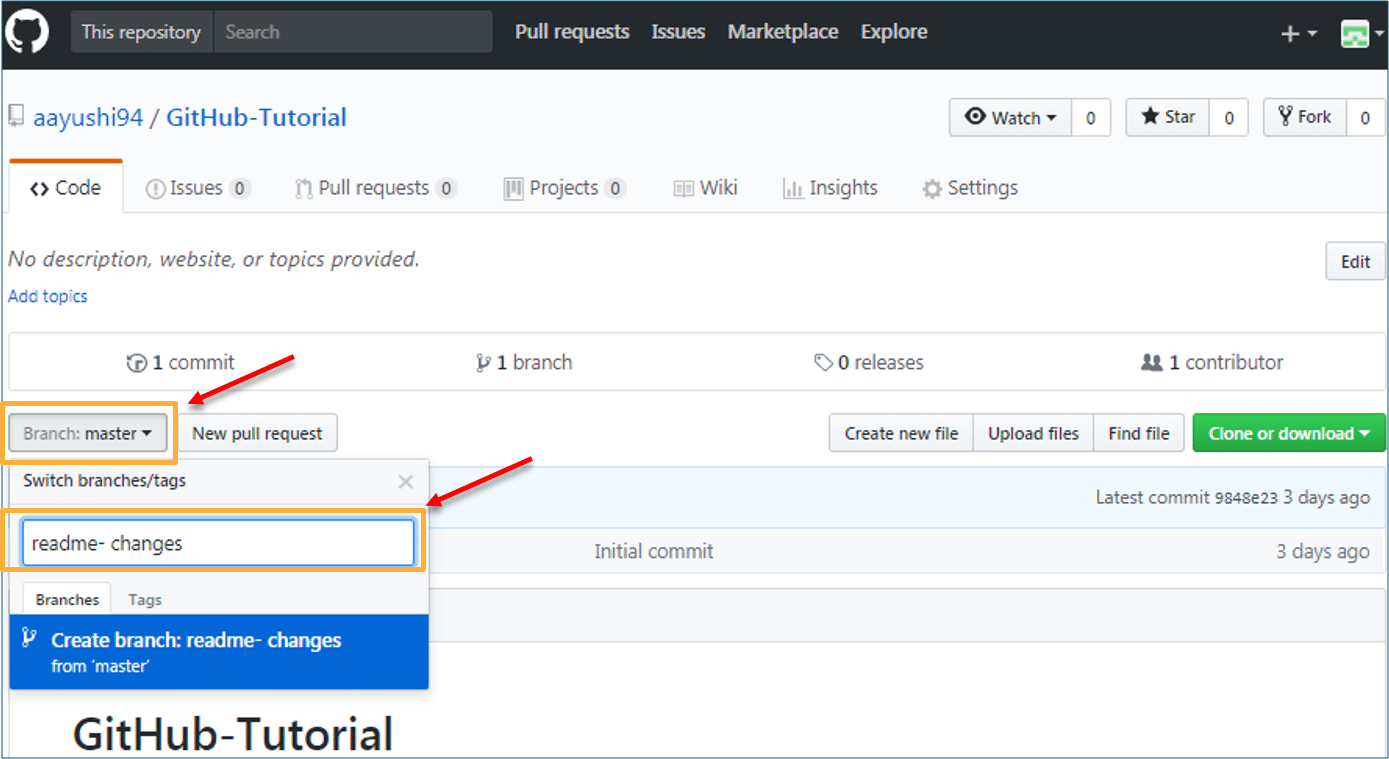
## ****Step 3: Create Branches and Perform Operations****

**Branching:**Branches help you to work on different versions of a repository at one time. Let’s say you want to add a new feature (which is in the development phase), and you are afraid at the same time whether to make changes to your main project or not. This is where git branching comes to rescue. Branches allow you to move back and forth between the different states/versions of a project. In the above scenario, you can create a new branch and test the new feature without affecting the main branch. Once you are done with it, you can merge the changes from new branch to the main branch. Here the main branch is the master branch, which is there in your repository by default. Refer to the below image for better understanding:

As depicted in the above image, there is a master/ production branch which has a new branch for testing. Under this branch, two set of changes are done and once it completed, it is merged back to the master branch. So this is how branching works!  
Let’s move ahead in ‘how to use GitHub’ blog, and learn how you can create a branch.

To create a branch in GitHub, follow the below steps:

* Click on the dropdown “Branch: master”
* As soon as you click on the branch, you can find an existing branch or you can create a new one. In my case, I am creating a new branch with a name “readme- changes”. Refer to the below screenshot for better understanding.



Once you have created a new branch, you have two branches in your repository now i.e. read-me (master branch) and readme- changes. The new branch is just the copy of master branch. So let’s perform some changes in our new branch and make it look different from the master branch.

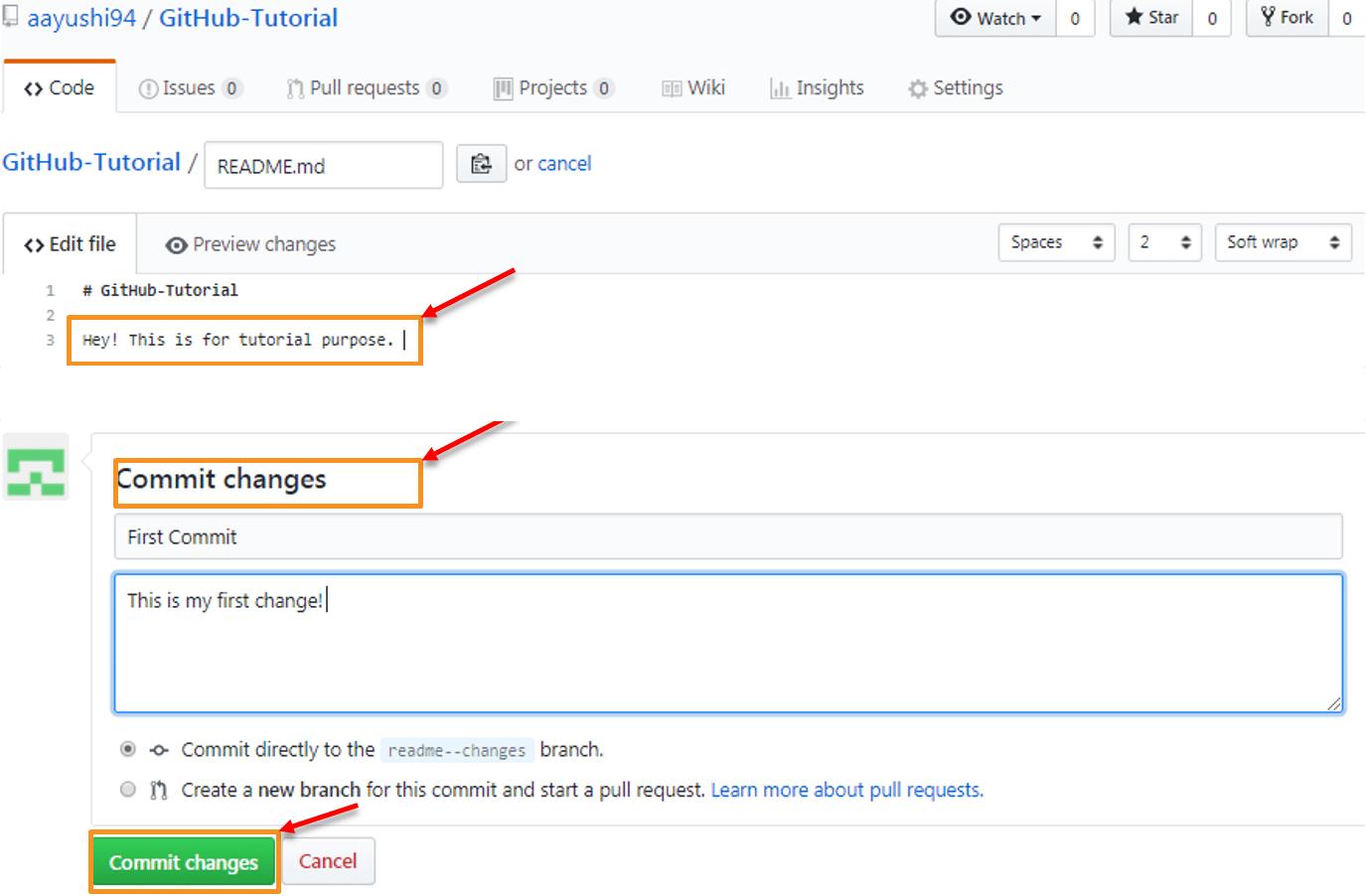
### **How to use GitHub: Operations**

#### **Commit Command:**

This operation helps you to save the changes in your file. When you commit a file, you should always provide the message, just to keep in the mind the changes done by you. Though this message is not compulsory but it is always recommended so that it can differentiate the various versions or commits you have done so far to your repository. These commit messages maintain the history of changes which in turn help other contributors to understand the file better. Now let’s make our first commit, follow the below steps:

* Click on “readme- changes” file which we have just created.
* Click on the “edit” or a pencil icon in the righmost corner of the file.
* Once you click on that, an editor will open where you can type in the changes or anything.
* Write a commit message which identifies your changes.
* Click commit changes in the end.

Refer to the below screenshot for better understanding:

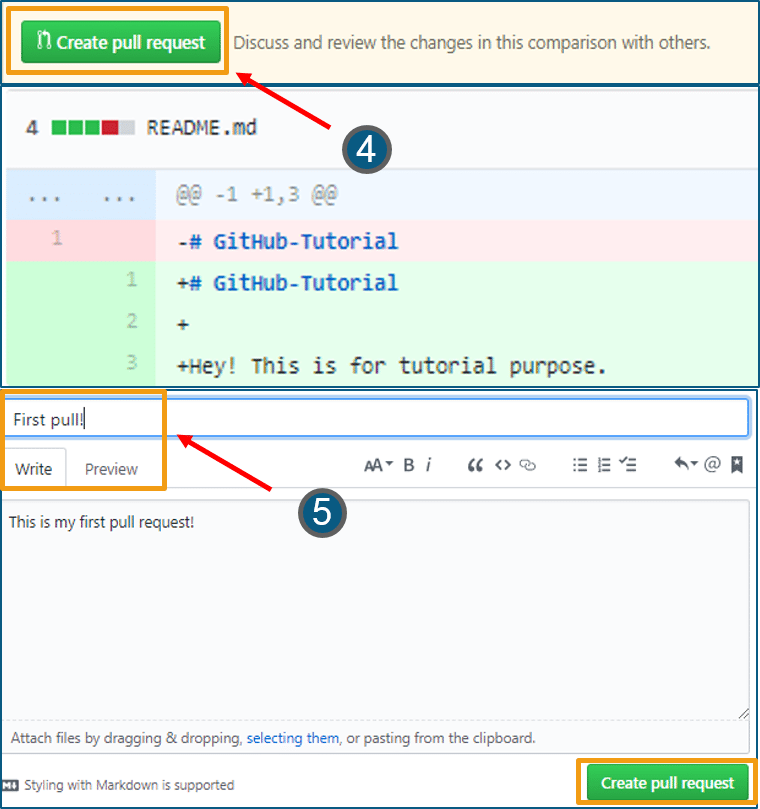
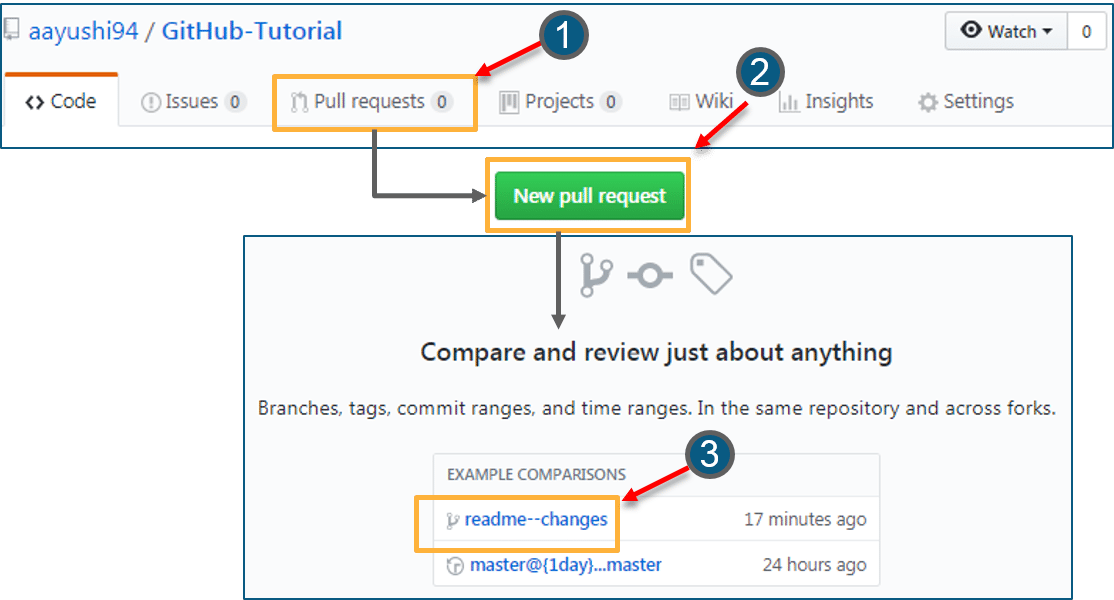


We have successfully made our first commit. Now this “readme- changes” file is different from the master branch. Next, let us see how can we open a pull request.

#### **Pull Command**

Pull command is the most important command in GitHub. It tell the changes done in the file and request other contributors to view it as well as merge it with the master branch. Once the commit is done, anyone can pull the file and can start a discussion over it. Once its all done, you can merge the file. Pull command compares the changes which are done in the file and if there are any conflicts, you can manually resolve it. Now let us see different steps involved to pull request in GitHub.

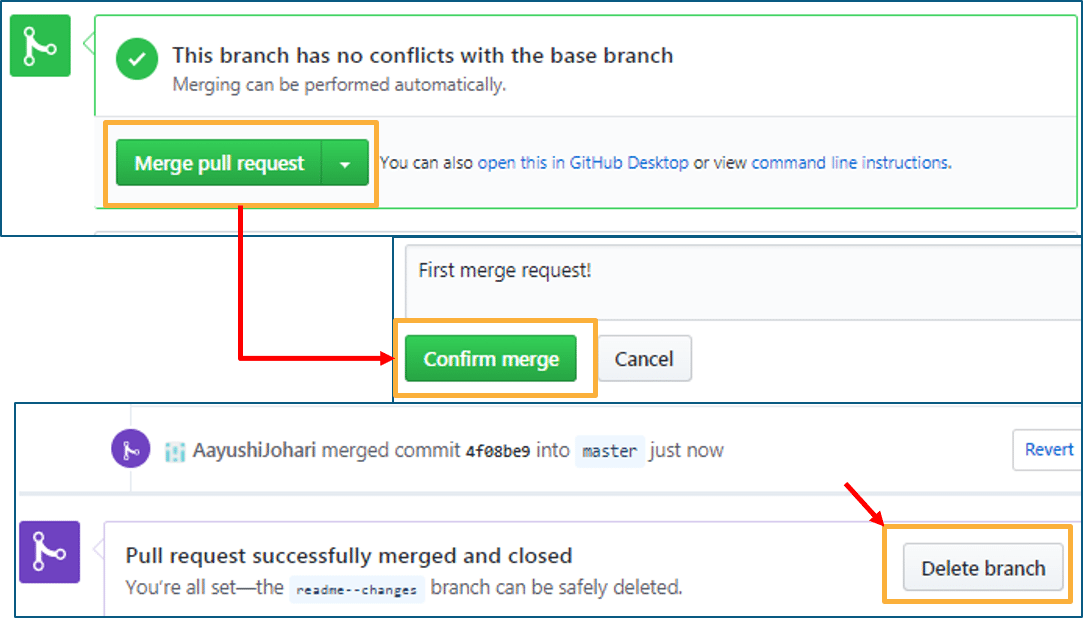
* Click the ‘Pull requests’ tab.
* Click ‘New pull request’.
* Once you click on pull request, select the branch and click ‘readme- changes’ file to view changes between the two files present in our repository.
* Click “Create pull request”.
* Enter any title, description to your changes and click on “Create pull request”. Refer to the below screenshots.

Next, let us move forward and see how can you merge your pull request.

#### **Merge Command**

Here comes the last command which merge the changes into the main master branch. We saw the changes in pink and green color, now let’s merge the “readme- changes” file with the master branch/ read-me. Go through the below steps to merge pull request.

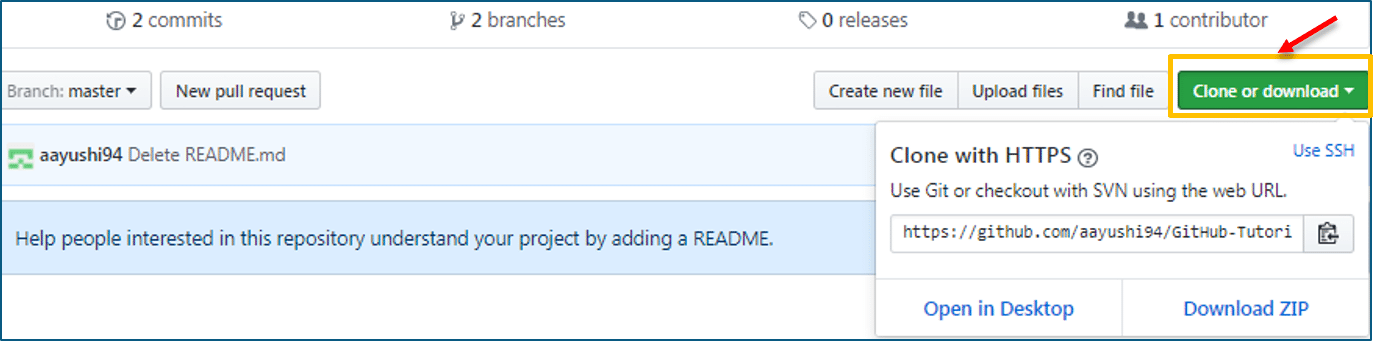
* Click on “Merge pull request” to merge the changes into master branch.
* Click “Confirm merge”.
* You can delete the branch once all the changes have been incorporated and if there are no conflicts. Refer to the below screenshots.



I hope you guys are trying these steps simultaneously while you are learning how to use GitHub. Next, let us move to our last topic in ‘how to use GitHub’ blog, i.e. Cloning and forking a GitHub repository.

## ****Step 4: Cloning and Forking GitHub Repository****

**Cloning:** Before I actually talk about cloning a GitHub repository, first let us understand why do we need to clone a repository. The answer is simple! Suppose you want to use some code which is present in a public repository, you can directly copy the contents by cloning or downloading. Refer to the below screenshot for a better understanding.



Cloning is really simple! In case you are facing any challenges on how to use GitHub, please comment your problems in the section below. Moving forward, let’s see what forking is.

**Forking:** First, let us talk about why do we need forking. Suppose, you need some code which is present in a public repository, under your repository and GitHub account. For this, we need to fork a repository.

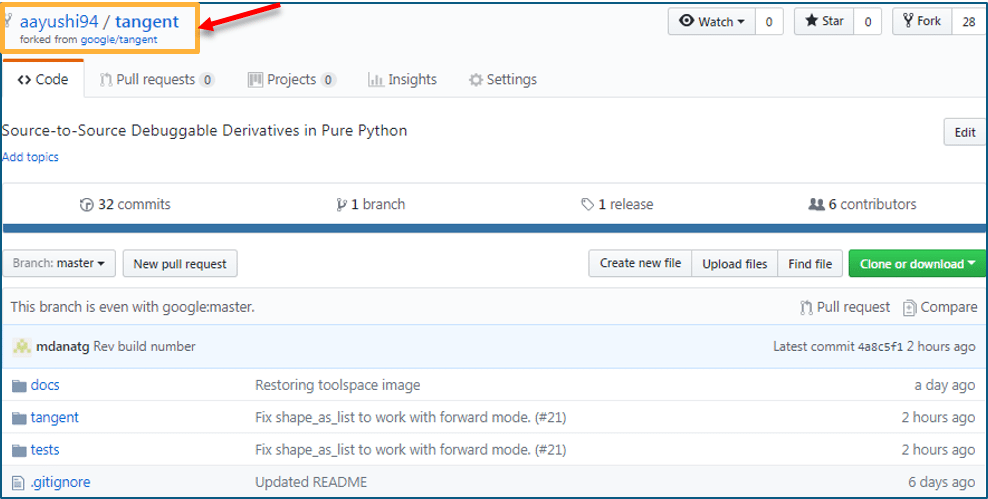
Before we get started with forking, there are some important points which you should always keep in mind.

* Changes done to the original repository will be reflected back to the forked repository.
* If you make a change in forked repository, it will not b reflected to the original repository until and unless you have made a pull request.

Now let’s see how can you want to fork a repository. For that, follow the below steps:

* Go to Explore and search for public repositories.
* Click “fork”. Note that this “tangent” repository is already forked 27 times and it is under “google”account. Refer the below image for better understanding.  
    
  

As soon as you click on “Fork”, it will take some time to fork the repository. Once done you will notice that the repository name is under your account. For reference, you can have a look at the below screenshot.



Congratulations! You have successfully forked an existing repository under your own account.

## Working with remote repositories

### **git remote**

To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository.

Usage:

# Add remote repository

$ git remote <command><remote\_name><remote\_URL>

# List named remote repositories

$ git remote -v

In Practice:

# Adding a remote repository with the name of beanstalk

$ git remote add origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

# List named remote repositories

$ git remote -v

origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git (fetch)

origin git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git (push)

Note: A remote repository can have any name. It’s common practice to name the remote repository ‘origin’.

### **git clone**

To create a local working copy of an existing remote repository, use git clone to copy and download the repository to a computer. Cloning is the equivalent of gitinit when working with a remote repository. Git will create a directory locally with all files and repository history.

Usage:

$ git clone <remote\_URL>

In Practice:

$ git clone git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

Cloning into 'repository\_name'...

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 5 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (5/5), 3.08 KiB | 0 bytes/s, done.

Checking connectivity... done.

### **git pull**

To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.

Usage:

$ git pull <branch\_name><remote\_URL/remote\_name>

In Practice:

# Pull from named remote

$ git pull origin staging

From account\_name.git.beanstalkapp.com:/account\_name/repository\_name

\* branch staging -> FETCH\_HEAD

\* [new branch] staging -> origin/staging

Already up-to-date.

# Pull from URL (not frequently used)

$ git pull git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git staging

From account\_name.git.beanstalkapp.com:/account\_name/repository\_name

\* branch staging -> FETCH\_HEAD

\* [new branch] staging -> origin/staging

Already up-to-date.

### **git push**

Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for.

Usage:

$ git push <remote\_URL/remote\_name><branch>

# Push all local branches to remote repository

$ git push —all

In Practice:

# Push a specific branch to a remote with named remote

$ git push origin staging

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (5/5), 734 bytes | 0 bytes/s, done.

Total 5 (delta 2), reused 0 (delta 0)

To git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

ad189cb..0254c3d SecretTesting ->SecretTesting

# Push all local branches to remote repository

$ git push --all

Counting objects: 4, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 373 bytes | 0 bytes/s, done.

Total 4 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 2 local objects.

To git@account\_name.git.beanstalkapp.com:/acccount\_name/repository\_name.git

0d56917..948ac97 master -> master

ad189cb..0254c3d SecretTesting ->SecretTesting

## Git fetch commands and options

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

git fetch --all

A power move which fetches all registered remotes and their branches:

git fetch --dry-run

The --dry-run option will perform a demo run of the command. I will output examples of actions it will take during the fetch but not apply them.

## Git fetch examples

## git fetch a remote branch

The following example will demonstrate how to fetch a remote branch and update your local working state to the remote contents. In this example, lets assume there is a central repo origin which the local repository has been cloned from using the git clone command. Let us also assume an additional remote repository named coworkers\_repo that contains a feature\_branch which we will configure and fetch. With these assumptions set let us continue the example.

Firstly we will need to configure the remote repo using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command.

git remote coworkers\_repo git@bitbucket.org:coworker/coworkers\_repo.git

Here we have created a reference to the coworker's repo using the repo URL. We will now pass that remote name to gitfetchto download the contents.

git fetch coworkers feature\_branch  
fetching coworkers/feature\_branch

We now locally have the contents of coworkers/feature\_branch we will need the integrate this into our local working copy. We begin this process by using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command to checkout the newly downloaded remote branch.

git checkout coworkers/feature\_branch  
Note: checking out coworkers/feature\_branch'.  
  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by performing another checkout.  
  
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:  
  
git checkout -b <new-branch-name>

The output from this checkout operation indicates that we are in a detached HEAD state. This is expected and means that our HEADref is pointing to a ref that is not in sequence with our local history. Being that HEAD is pointed at the coworkers/feature\_branch ref, we can create a new local branch from that ref. The 'detached HEAD' output shows us how to do this using the git checkout command:

git checkout -b local\_feature\_branch

Here we have created a new local branch named local\_feature\_branch this puts updates HEAD to point at the latest remote content and we can continue development on it from this point.

## Synchronize origin with git fetch

The following example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

git fetch origin

This will display the branches that were downloaded:

a1e8fb5..45e66a4 master -> origin/master  
a1e8fb5..9e8ab1c develop -> origin/develop  
\* [new branch] some-feature -> origin/some-feature

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.

To see what commits have been added to the upstream master, you can run a git log using origin/master as a filter:

git log --oneline master..origin/master

To approve the changes and merge them into your local master branch with the following commands:

git checkout master  
git log origin/master

Then we can use git merge origin/master:

git merge origin/master

The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

## Git fetch summary

In review, git fetch is a primary command used to download contents from a remote repository. git fetch is used in conjunction with git remote, git branch, git checkout, and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to update a local repository to the state of a remote. The git fetch command is a critical piece of collaborative git work flows. git fetch has similar behavior to git pull however git fetch can be considered a safer, nondestructive version.

**Handling Git Conflicts:**

Jerry is working on the **wchar\_support** branch. He changes the name of the functions and after testing, he commits his changes.

[jerry@CentOSsrc]$ git branch

master

\* wchar\_support

[jerry@CentOSsrc]$ git diff

The above command produces the following result:

diff --git a/src/string\_operations.c b/src/string\_operations.c

index 8fb4b00..01ff4e0 100644

--- a/src/string\_operations.c

+++ b/src/string\_operations.c

@@ -1,7 +1,7 @@

#include <stdio.h>

#include <wchar.h>

-size\_tw\_strlen(constwchar\_t \*s)

+size\_tmy\_wstrlen(constwchar\_t \*s)

{

constwchar\_t \*p = s;

After verifying the code he commits his changes.

[jerry@CentOSsrc]$ git status -s

M string\_operations.c

[jerry@CentOSsrc]$ git add string\_operations.c

[jerry@CentOSsrc]$ git commit -m 'Changed function name'

[wchar\_support 3789fe8] Changed function name

1 files changed, 1 insertions(+), 1 deletions(-)

[jerry@CentOSsrc]$ git push origin wchar\_support

The above command will produce the following result:

Counting objects: 7, done.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 409 bytes, done.

Total 4 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

64192f9..3789fe8 wchar\_support ->wchar\_support

## Perform Changes in Master Branch

Meanwhile in the master branch, Tom also changes the name of the same function and pushes his changes to the master branch.

[tom@CentOSsrc]$ git branch

\* master

[tom@CentOSsrc]$ git diff

The above command produces the following result:

diff --git a/src/string\_operations.c b/src/string\_operations.c

index 8fb4b00..52bec84 100644

--- a/src/string\_operations.c

+++ b/src/string\_operations.c

@@ -1,7 +1,8 @@

#include <stdio.h>

#include <wchar.h>

-size\_tw\_strlen(constwchar\_t \*s)

+/\* wide character strlenfucntion \*/

+size\_tmy\_wc\_strlen(constwchar\_t \*s)

{

constwchar\_t \*p = s;

After verifying diff, he commits his changes.

[tom@CentOSsrc]$ git status -s

M string\_operations.c

[tom@CentOSsrc]$ git add string\_operations.c

[tom@CentOSsrc]$ git commit -m 'Changed function name from w\_strlen to my\_wc\_strlen'

[master ad4b530] Changed function name from w\_strlen to my\_wc\_strlen

1 files changed, 2 insertions(+), 1 deletions(-)

[tom@CentOSsrc]$ git push origin master

The above command will produce the following result:

Counting objects: 7, done.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 470 bytes, done.

Total 4 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

64192f9..ad4b530 master -> master

On the **wchar\_support** branch, Jerry implements strchr function for wide character string. After testing, he commits and pushes his changes to the **wchar\_support** branch.

[jerry@CentOSsrc]$ git branch

master

\* wchar\_support

[jerry@CentOSsrc]$ git diff

The above command produces the following result:

diff --git a/src/string\_operations.c b/src/string\_operations.c

index 01ff4e0..163a779 100644

--- a/src/string\_operations.c

+++ b/src/string\_operations.c

@@ -1,6 +1,16 @@

#include <stdio.h>

#include <wchar.h>

+wchar\_t \*my\_wstrchr(wchar\_t \*ws, wchar\_twc)

+

{

+

while (\*ws)

{

+

if (\*ws == wc)

+

returnws;

+

++ws;

+

}

+ return NULL;

+

}

+

size\_tmy\_wstrlen(constwchar\_t \*s)

{

constwchar\_t \*p = s;

After verifying, he commits his changes.

[jerry@CentOSsrc]$ git status -s

M string\_operations.c

[jerry@CentOSsrc]$ git add string\_operations.c

[jerry@CentOSsrc]$ git commit -m 'Adddedstrchr function for wide character string'

[wchar\_support 9d201a9] Adddedstrchr function for wide character string

1 files changed, 10 insertions(+), 0 deletions(-)

[jerry@CentOSsrc]$ git push origin wchar\_support

The above command will produce the following result:

Counting objects: 7, done.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 516 bytes, done.

Total 4 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

3789fe8..9d201a9 wchar\_support ->wchar\_support

## Tackle Conflicts

Tom wants to see what Jerry is doing on his private branch so, he tries to pull the latest changes from the **wchar\_support** branch, but Git aborts the operation with the following error message.

[tom@CentOSsrc]$ git pull origin wchar\_support

The above command produces the following result:

remote: Counting objects: 11, done.

63Git Tutorials

remote: Compressing objects: 100% (8/8), done.

remote: Total 8 (delta 2), reused 0 (delta 0)

Unpacking objects: 100% (8/8), done.

From git.server.com:project

\* branch

wchar\_support -> FETCH\_HEAD

Auto-merging src/string\_operations.c

**CONFLICT (content): Merge conflict in src/string\_operations.c**

Automatic merge failed; fix conflicts and then commit the result.

## Resolve Conflicts

From the error message, it is clear that there is a conflict in src/string\_operations.c . He runs the git diff command to view further details.

[tom@CentOSsrc]$ git diff

The above command produces the following result:

diff --cc src/string\_operations.c

index 52bec84,163a779..0000000

--- a/src/string\_operations.c

+++ b/src/string\_operations.c

@@@ -1,8 -1,17 +1,22 @@@

#include <stdio.h>

#include <wchar.h>

++<<<<<<< HEAD

+/\* wide character strlenfucntion \*/

+size\_tmy\_wc\_strlen(constwchar\_t \*s)

++=======

+ wchar\_t \*my\_wstrchr(wchar\_t \*ws, wchar\_twc)

+

{

+

+

while (\*ws)

{

if (\*ws == wc)

+

returnws;

+

++ws;

+

}

+ return NULL;

+

}

+

+ size\_tmy\_wstrlen(constwchar\_t \*s)

++>>>>>>>9d201a9c61bc4713f4095175f8954b642dae8f86

{

constwchar\_t \*p = s;

As both Tom and Jerry changed the name of the same function, Git is in a state of confusion and it asks the user to resolve the conflict manually.

Tom decides to keep the function name suggested by Jerry, but he keeps the comment added by him, as it is. After removing the conflict markers, git diff will look like this.

[tom@CentOSsrc]$ git diff

The above command produces the following result.

diff --cc src/string\_operations.c

diff --cc src/string\_operations.c

index 52bec84,163a779..0000000

--- a/src/string\_operations.c

+++ b/src/string\_operations.c

@@@ -1,8 -1,17 +1,18 @@@

#include <stdio.h>

#include <wchar.h>

+ wchar\_t \*my\_wstrchr(wchar\_t \*ws, wchar\_twc)

+

{

+

while (\*ws)

{

+

if (\*ws == wc)

+

returnws;

+

++ws;

+

}

+ return NULL;

+

}

+

+/\* wide character strlenfucntion \*/

- size\_tmy\_wc\_strlen(constwchar\_t \*s)

+ size\_tmy\_wstrlen(constwchar\_t \*s)

{

constwchar\_t \*p = s;

As Tom has modified the files, he has to commit these changes first and thereafter, he can pull the changes.

[tom@CentOSsrc]$ git commit -a -m 'Resolved conflict'

[master 6b1ac36] Resolved conflict

[tom@CentOSsrc]$ git pull origin wchar\_support.

Tom has resolved the conflict, now the pull operation will succeed.

**Git Interview Questions And Answers:**

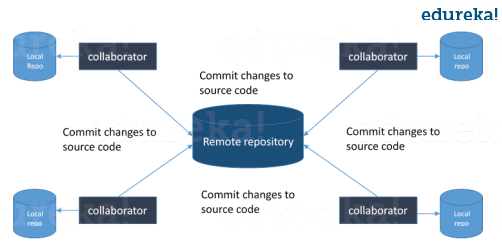
**Q1. What is the difference between Git and SVN?**

|  |  |
| --- | --- |
| **Git vs SVN** | |
| **Git** | **SVN** |
| 1. Git is a Decentralized Version Control tool | 1. SVN is a  Centralized Version Control tool |
| 2.  It belongs to the 3rd generation of Version Control tools | 2. It belongs to the 2nd generation of Version Control tools |
| 3. Clients can clone entire repositories on their local systems | 3.  Version history is stored on a server-side repository |
| 4. Commits are possible even if offline | 4. Only online commits are allowed |
| 5. Push/pull operations are faster | 5. Push/pull operations are slower |
| 6. Works are shared automatically by commit | 6. Nothing is shared automatically |

**Q2. What is Git?**

I will suggest you to attempt this question by first telling about the architecture of git as shown in the below diagram just try to explain the diagram by saying:

* Git is a Distributed Version Control system (DVCS). It can track changes to a file and allows you to revert back to any particular change.
* Its distributed architecture provides many advantages over other Version Control Systems (VCS) like SVN one major advantage is that it does not rely on a central server to store all the versions of a project’s files.
* Instead, every developer “clones” a copy of a repository I have shown in the diagram with “Local repository” and has the full history of the project on his hard drive so when there is a server outage all you need for recovery is one of your teammate’s local Git repository.
* There is a central cloud repository as well where developers can commit changes and share it with other teammates as you can see in the diagram where all collaborators are commiting changes “Remote repository”.



Now, the next set of Git interview questions will test your experience with Git:

**Q3. What is the command to write a commit message in Git?**

Answer to this is pretty straightforward.

Command that is used to write a commit message is “**git commit -a**”.

Now explain about -a flag by saying -a on the command line instructs git to commit the new content of all tracked files that have been modified. Also mention you can use “**git add<file>**” before git commit -a if new files need to be committed for the first time.

**Q4. What is ‘bare repository’ in Git?**

You are expected to tell the difference between a “working directory” and “bare repository”.

A “bare” repository in Git just contains the version control information and no working files (no tree) and it doesn’t contain the special .git sub-directory. Instead, it contains all the contents of the .git sub-directory directly in the main directory itself, where as working directory consist of:

1. A .git subdirectory with all the Git related revision history of your repo.
2. A working tree, or checked out copies of your project files.

**Q5. What language is used in Git?**

Instead of just telling the name of the language, you need to tell the reason for using it as well. I will suggest you to answer this by saying:

Git uses ‘C’ language. GIT is fast, and ‘C’ language makes this possible by reducing the overhead of run times associated with high level languages.

[**Learn GitWith DevOps Now**](https://www.edureka.co/devops/)

**Q6. In Git how do you revert a commit that has already been pushed and made public?**

There can be two answers to this question and make sure that you include both because any of the below options can be used depending on the situation:

* Remove or fix the bad file in a new commit and push it to the remote repository. This is the most natural way to fix an error. Once you have made necessary changes to the file, commit it to the remote repository for that I will use  
  **git commit -m “commit message”**
* Create a new commit that undoes all changes that were made in the bad commit.to do this I will use a command  
  **git revert <name of bad commit>**

**Q7. What is the difference between git pull and git fetch?**

Git pull command pulls new changes or commits from a particular branch from your central repository and updates your target branch in your local repository.

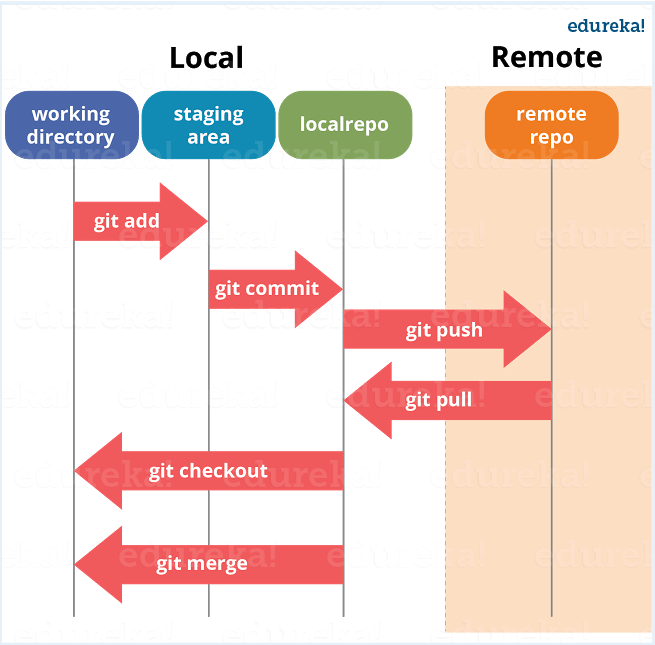
Git fetch is also used for the same purpose but it works in a slightly different way. When you perform a git fetch, it pulls all new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, git fetch must be followed with a git merge. Your target branch will only be updated after merging the target branch and fetched branch. Just to make it easy for you, remember the equation below:

Git pull = git fetch + git merge

**Q8. What is ‘staging area’ or ‘index’ in Git?**

For this answer try to explain the below diagram as you can see:

That before completing the commits, it can be formatted and reviewed in an intermediate area known as ‘Staging Area’ or ‘Index’. From the diagram it is evident that every change is first verified in the staging area I have termed it as “stage file” and then that change is committed to the repository.



If your interviewer has good knowledge on Git he/she will dig in deep, so the next set of Git interview questions will be more challenging.

**Q9. What is Git stash?**

According to me you should first explain the need for Git stash.

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for sometime to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is Git stash.

Now explain what is Git stash.

Stashing takes your working directory that is, your modified tracked files and staged changes and saves it on a stack of unfinished changes that you can reapply at any time.

**Q10. What is Git stash drop?**

Begin this answer by saying for what purpose we use Git ‘stash drop’.

Git ‘stash drop’ command is used to remove the stashed item. It will remove the last added stash item by default, and it can also remove a specific item if you include it as an argument.

Now give an example.

If you want to remove a particular stash item from the list of stashed items you can use the below commands:

**git stash list:**It will display the list of stashed items like:  
stash@{0}: WIP on master: 049d078 added the index file  
stash@{1}: WIP on master: c264051 Revert “added file\_size”  
stash@{2}: WIP on master: 21d80a5 added number to log

If you want to remove an item named stash@{0} use command **git stash drop stash@{0}**.

**Q11. How do you find a list of files that has changed in a particular commit?**

For this answer instead of just telling the command, explain what exactly this command will do.

To get a list files that has changed in a particular commit use the below command:

**git diff-tree -r {hash}**

Given the commit hash, this will list all the files that were changed or added in that commit. The -r flag makes the command list individual files, rather than collapsing them into root directory names only.

You can also include the below mentioned point, although it is totally optional but will help in impressing the interviewer.

The output will also include some extra information, which can be easily suppressed by including two flags:

**git diff-tree –no-commit-id –name-only -r {hash}**

Here –no-commit-id will suppress the commit hashes from appearing in the output, and –name-only will only print the file names, instead of their paths.

**Q12. What is the function of ‘gitconfig’?**

First tell why we need ‘**gitconfig**‘.

Git uses your username to associate commits with an identity. The gitconfig command can be used to change your Git configuration, including your username.

Now explain with an example.

Suppose you want to give a username and email id to associate commit with an identity so that you can know who has made a particular commit. For that I will use:

**gitconfig –global user.name “Your Name”:**This command will add username.  
**gitconfig –global user.email “Your E-mail Address”:**This command will add email id.

**Q13. What does commit object contains?**

Commit object contains the following components, you should mention all the three points present below:

* A set of files, representing the state of a project at a given point of time
* Reference to parent commit objects
* An SHAI name, a 40 character string that uniquely identifies the commit object.

**Q14. How can you create a repository in Git?**

This is probably the most frequently asked questions and answer to this is really simple.

To create a repository, create a directory for the project if it does not exist, then run command “**gitinit**”. By running this command .git directory will be created in the project directory.

**Q15. How do you squash last N commits into a single commit?**

There are two options to squash last N commits into a single commit include both of the below mentioned options in your answer:

* If you want to write the new commit message from scratch use the following command  
  **gitreset –soft HEAD~N &&**  
  **git commit**
* If you want to start editing the new commit message with a concatenation of the existing commit messages then you need to extract those messages and pass them to Git commit for that I will use  
  **git reset –soft HEAD~N &&**  
  **git commit –edit -m”$(git log –format=%B –reverse**[**.HEAD@{N}**](mailto:HEAD..HEAD@%7b1%7d)**)”**

**Q16. What is Git bisect? How can you use it to determine the source of a (regression) bug?**

I will suggest you to first give a small definition of Git bisect.

Git bisect is used to find the commit that introduced a bug by using binary search. Command for Git bisect is  
**git bisect <subcommand><options>**

Now since you have mentioned the command above explain them what this command will do.

This command uses a binary search algorithm to find which commit in your project’s history introduced a bug. You use it by first telling it a “bad” commit that is known to contain the bug, and a “good” commit that is known to be before the bug was introduced. Then Git bisect picks a commit between those two endpoints and asks you whether the selected commit is “good” or “bad”. It continues narrowing down the range until it finds the exact commit that introduced the change.

**Q17. How do you configure a Git repository to run code sanity checking tools right before making commits, and preventing them if the test fails?**

I will suggest you to first give a small introduction to sanity checking.

A sanity or smoke testdetermines whether it is possible and reasonable to continue testing.

Now explain how to achieve this.

This can be done with a simple script related to the pre-commit hook of the repository. The pre-commit hook is triggered right before a commit is made, even before you are required to enter a commit message. In this script one can run other tools, such as linters and perform sanity checks on the changes being committed into the repository.

Finally, give an example, you can refer the below script:

**#!/bin/sh**  
**files=$(git diff –cached –name-only –diff-filter=ACM | grep ‘.go$’)**  
**if[ -z files ]; then**  
**exit 0**  
**fi**  
**unfmtd=$(gofmt -l $files)**  
**if [ -z unfmtd ]; then**  
**exit 0**  
**fi**  
**echo “Some .go files are not fmt’d”**  
**exit 1**This script checks to see if any .go file that is about to be committed needs to be passed through the standard Go source code formatting tool gofmt. By exiting with a non-zero status, the script effectively prevents the commit from being applied to the repository.

The Interviewer has not started asking questions on branching yet, so the next set of Git interview questions will be dealing with branching in Git.

**Q18. Describe branching strategies you have used?**

This question is asked to test your branching experience with Git so, tell them about how you have used branching in your previous job and what purpose does it serves, you can refer the below mention points:

* Feature branching  
  A feature branch model keeps all of the changes for a particular feature inside of a branch. When the feature is fully tested and validated by automated tests, the branch is then merged into master.
* Task branching  
  In this model each task is implemented on its own branch with the task key included in the branch name. It is easy to see which code implements which task, just look for the task key in the branch name.
* Release branching  
  Once the develop branch has acquired enough features for a release, you can clone that branch to form a Release branch. Creating this branch starts the next release cycle, so no new features can be added after this point, only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it is ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into develop branch, which may have progressed since the release was initiated.

In the end tell them that branching strategies varies from one organization to another so I know basic branching operations like delete, merge, checking out a branch etc..

**Q19. How will you know in Git if a branch has already been merged into master?**

The answer is pretty direct.

To know if a branch has been merged into master or not you can use the below commands:

**git branch –merged** It lists the branches that have been merged into the current branch.  
**git branch –no-merged** It lists the branches that have not been merged.

**Q20. What is Git rebase and how can it be used to resolve conflicts in a feature branch before merge?**

According to me you should start by saying git rebase is a command which will merge another branch into the branch where you are currently working, and move all of the local commits that are ahead of the rebased branch to the top of the history on that branch.

Now, once you have defined Git rebase time for an example to show how it can be used to resolve conflicts in a feature branch before merge.

If a feature branch was created from the master, and since then the master branch has received new commits, Git rebase can be used to move the feature branch to the tip of master. The command effectively will replay the changes made in the feature branch at the tip of master, allowing conflicts to be resolved in the process. When done with care, this will allow the feature branch to be merged into master with relative ease and sometimes as a simple fast-forward operation.

You can also expect some off track questions, so the next question in this Git interview questions blog will be regarding SubGit.

**Q21. What is SubGit?**

Begin this answer by explaining what is SubGit used for.

SubGit is a tool for SVN to Git migration. It creates a writable Git mirror of a local or remote Subversion repository and uses both Subversion and Git as long as you like.

Now you can include some advantages like you can do a fast one-time import from Subversion to Git or use SubGit within AtlassianBitbucketServer.We can use SubGit to create a bi-directional Git-SVN mirror of existing Subversion repository. You can push to Git or commit to Subversion at your convenience. Synchronization will be done by SubGit.