GIT

Git is a free, open source distributed version control system tool designed to handle everything from small to very large projects with speed and efficiency. It was created by Linus Torvalds in 2005 to develop Linux Kernel. Git has the functionality, performance, security and flexibility that most teams and individual developers need. This 'What Is Git' blog is the first blog of my Git Tutorial series. I hope you will enjoy it. :-)

In this 'What is Git' blog, you will learn:

What is Git – Why Git Came Into Existence?

We all know "Necessity is the mother of all inventions". And similarly Git was also invented to fulfill certain necessities that the developers faced before Git. So, let us take a step back to learn all about Version Control Systems (VCS) and how Git came into existence.

Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

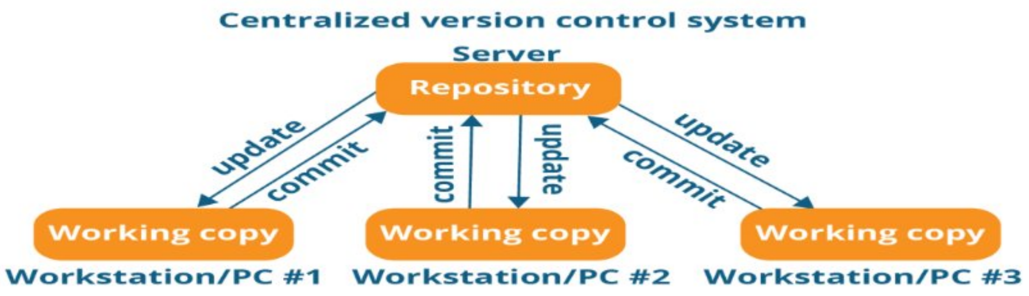
There are two types of VCS:

- 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.

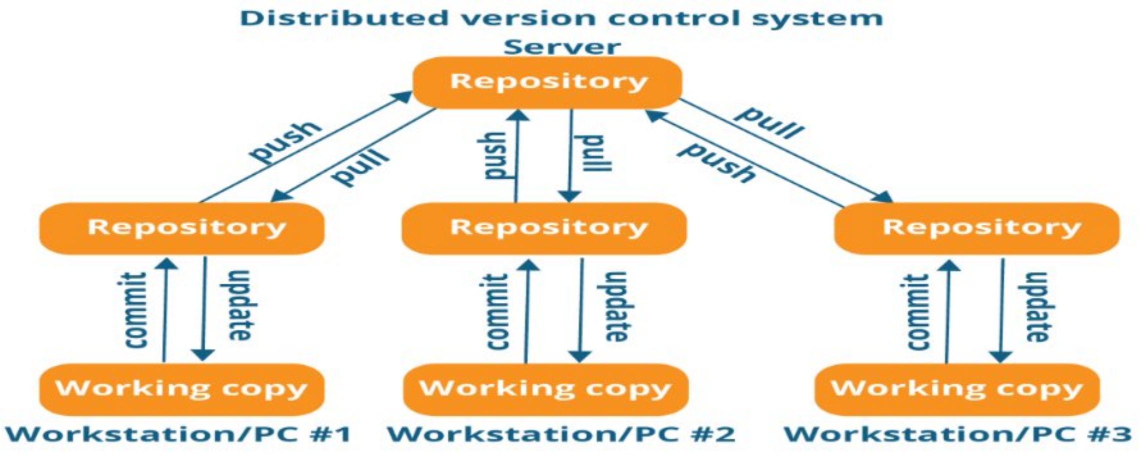
This is when Distributed VCS comes to the rescue.

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.

After knowing Distributed VCS, its time we take a dive into what is Git.

What is Git – Features Of Git

Freeandopensource:
Git is released under GPL's (General Public License) open source license. You don't need to purchase Git. It is absolutely free. And since it is open source, you can modify the source code as per your requirement.

Speed:
Since you do not have to connect to any network for performing all operations, it completes all the tasks really fast. Performance tests done by Mozilla showed it was an order of magnitude faster than other version control systems. Fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server. The core part of Git is written in C, which avoids runtime overheads associated with other high level languages.

Scalable:
Git is very scalable. So, if in future , the number of collaborators increase Git can easily handle this change. Though Git represents an entire repository, the data stored on the client's side is very small as Git compresses all the huge data through a lossless compressiontechnique.

Reliable:
Since every contributor has its own local repository, on the events of a system crash, the lost data can be recovered from any of the local repositories. You will always have a backup of all your files.

Secure:
Git uses the *SHA1* (Secure Hash Function) to name and identify objects within its repository. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.

Economical:
In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

Supports-non-linear-development:
Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in Git are very lightweight. A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

Easy_Branching:
Branch management with Git is very simple. It takes only few seconds to create, delete, and merge branches. Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something, no matter how big or small, they create a new branch. This ensures that the master branch always contains production-quality code.

Distributed_development:
Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

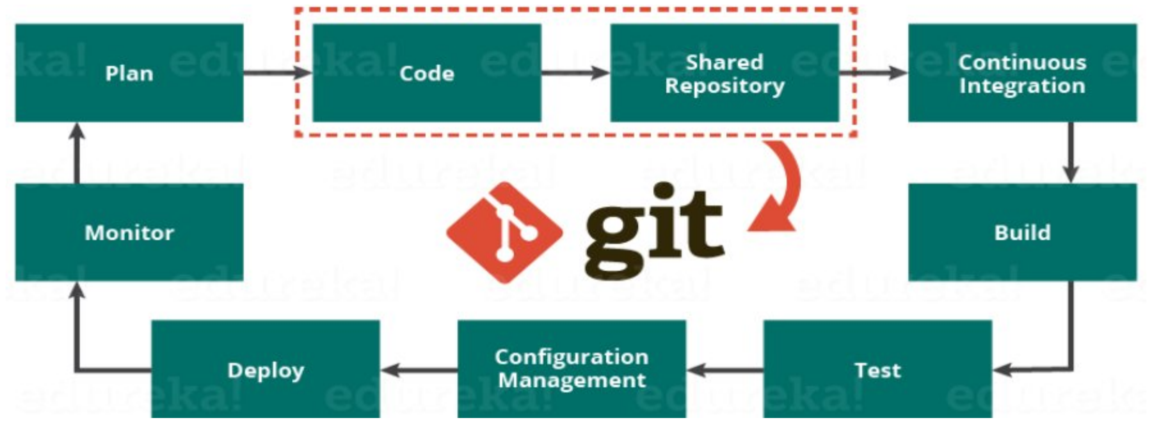
Compatibility-with-existing-systems-or-protocol
Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

What is Git – Role Of Git In DevOps?

Now that you know what is Git, you should know Git is an integral part of DevOps.

DevOps is the practice of bringing agility to the process of development and operations. It's an entirely new ideology that has swept IT organizations worldwide, boosting project life-cycles and in turn increasing profits. DevOps promotes communication between development engineers and operations, participating together in the entire service life-cycle, from design through the development process to production support.

The diagram below depicts the Devops life cycle and displays how Git fits in Devops.

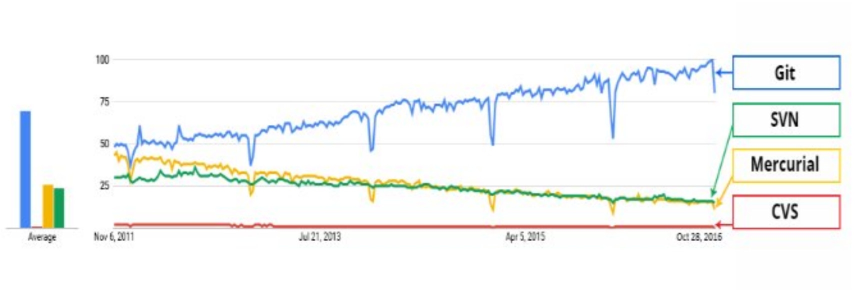


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration to create a build and test it on the test server and eventually deploy it on the production.

Tools like Git enable communication between the development and the operations team. When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project. Commit messages in Git play a very important role in communicating among the team. The bits and pieces that we all deploy lies in the Version Control system like Git. To succeed in DevOps, you need to have all of the communication in Version Control. Hence, Git plays a vital role in succeeding at DevOps.

Companies Using Git

Git has earned way more popularity compared to other version control tools available in the market like Apache Subversion(SVN), Concurrent Version Systems(CVS), Mercurial etc. You can compare the interest of Git by time with other version control tools with the graph collected from *Google Trends* below:



In large companies, products are generally developed by developers located all around the world. To enable communication among them, Git is the solution.

Some companies that use Git for version control are: Facebook, Yahoo, Zynga, Quora, Twitter, eBay, Salesforce, Microsoft and many more.

Lately, all of Microsoft's new development work has been in Git features. Microsoft is migrating .NET and many of its open source projects on GitHub which are managed by Git.

One of such projects is the LightGBM. It is a fast, distributed, high performance gradient boosting framework based on decision tree algorithms which is used for ranking, classification and many other machine learning tasks.

Here, Git plays an important role in managing this distributed version of LightGBM by providing speed and accuracy.

Git Tutorial – Operations & Commands

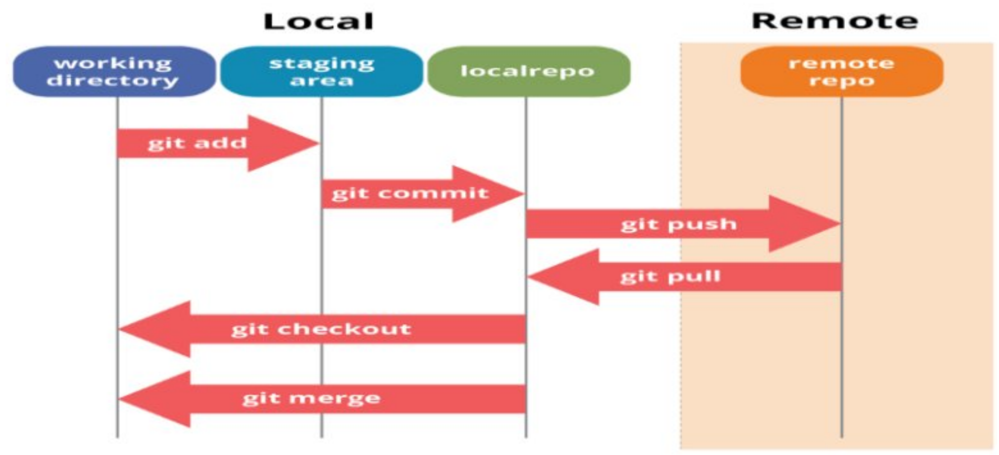
Some of the basic operations in Git are:

1. Initialize
2. Add
3. Commit
4. Pull
5. Push

Some advanced Git operations are:

1. Branching
2. Merging
3. Rebasing

Let me first give you a brief idea about how these operations work with the Git repositories. Take a look at the architecture of Git below:

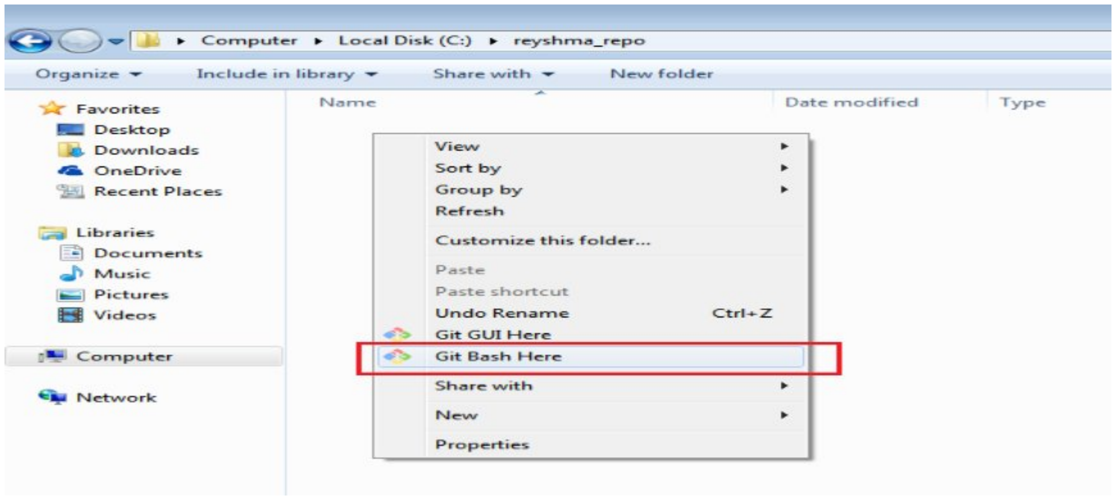


If you understand the above diagram well and good, but if you don't, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

You need to install Git on your system first. If you need help with the installation, *click here*.

In this Git Tutorial, I will show you the commands and the operations using Git Bash. Git Bash is a text-only command line interface for using Git on Windows which provides features to run automated scripts.

After installing Git in your Windows system, just open your folder/directory where you want to store all your project files; right click and select '*Git Bash here*'.

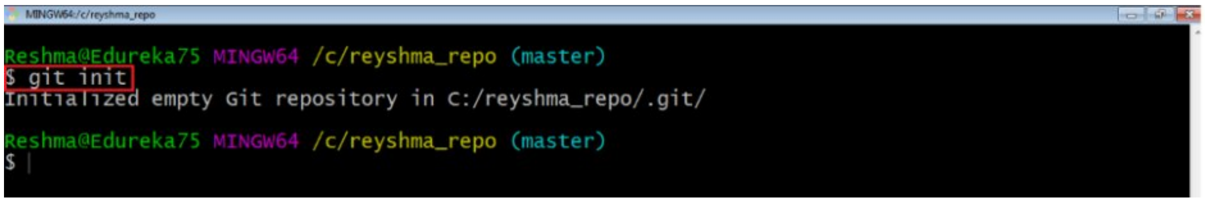


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

Now, the next task is to initialize your repository.

Initialize

In order to do that, we use the command git init. Please refer to the below screenshot.



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git init
Initialized empty Git repository in C:/reyshma_repo/.git/

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

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.

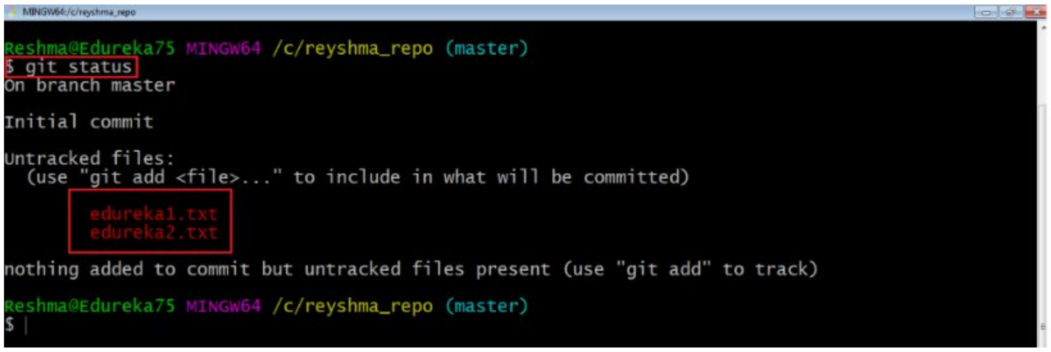
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely *edureka1.txt* and *edureka2.txt*.

Let's see if these files are in my index or not using the command git status. The index holds a snapshot of the content of the working tree/directory, and this snapshot is taken as the contents for the next change to be made in the local repository.

Git status

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

Let us type in the command to see what happens:



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git status
On branch master

Initial commit

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

        edureka1.txt
        edureka2.txt

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

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

This shows that I have two files which are not added to the index yet. This means I cannot commit changes with these files unless I have added them explicitly in the index.

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.

Thus, after making changes to the working tree, and before running the commit command, you must use the addcommand to add any new or modified files to the index. For that, use the commands below:

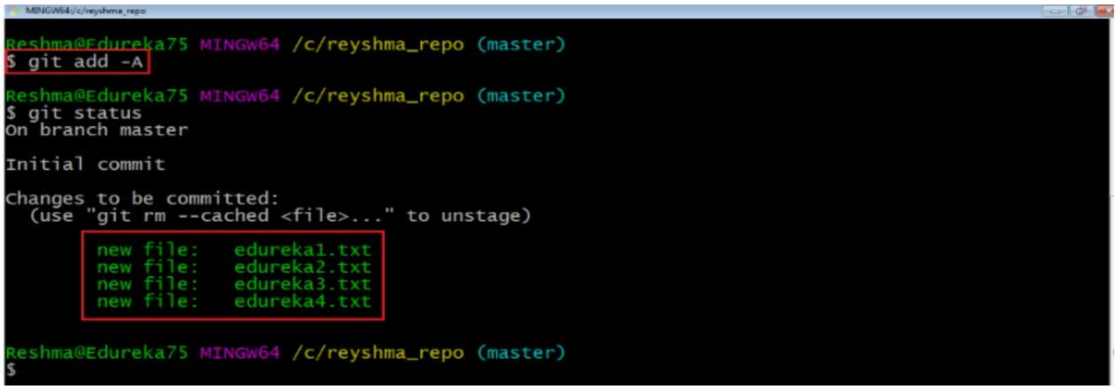
git add <directory>

or

git add <file>

Let me demonstrate the git add for you so that you can understand it better.

I have created two more files *edureka3.txt* and *edureka4.txt*. Let us add the files using the command git add -A. This command will add all the files to the index which are in the directory but not updated in the index yet.

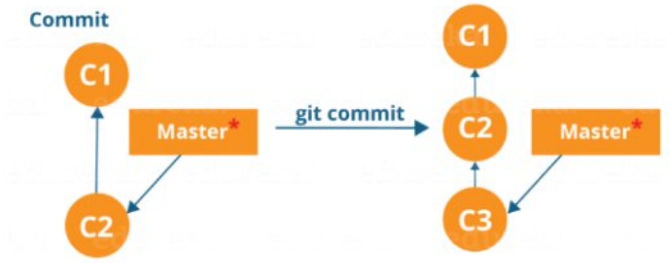


```
MINGW64:/c/reyshma_repo
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git add -A

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   edureka1.txt
        new file:   edureka2.txt
        new file:   edureka3.txt
        new file:   edureka4.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Now that the new files are added to the index, you are ready to commit them.

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



diagrambelow:

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.

You can commit by using the command below:

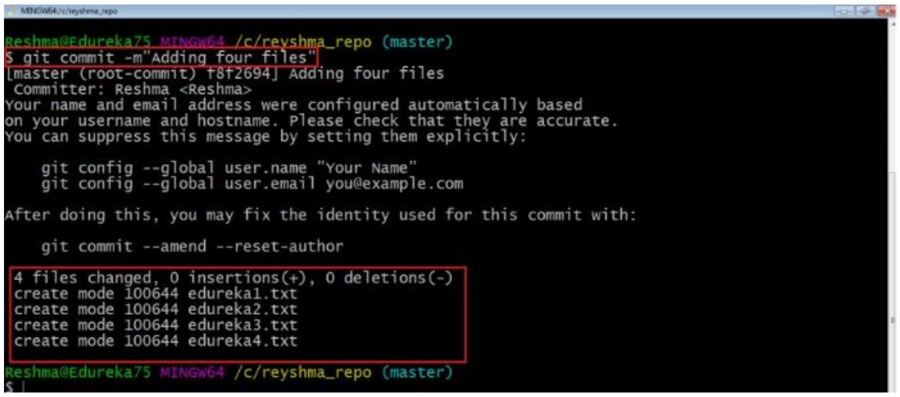
git commit

This will commit the staged snapshot and will launch a text editor prompting you for a commit message.

Or you can use:

git commit -m "<message>"

Let's try it out.



```
MINGW64:/c/reyshma_repo
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git commit -m"Adding four files"
[master (root-commit) f8f2694] Adding four files
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka1.txt
 create mode 100644 edureka2.txt
 create mode 100644 edureka3.txt
 create mode 100644 edureka4.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

As you can see above, the git commit command has committed the changes in the four files in the local repository.

Now, if you want to commit a snapshot of all the changes in the working directory at once, you can use the command below:

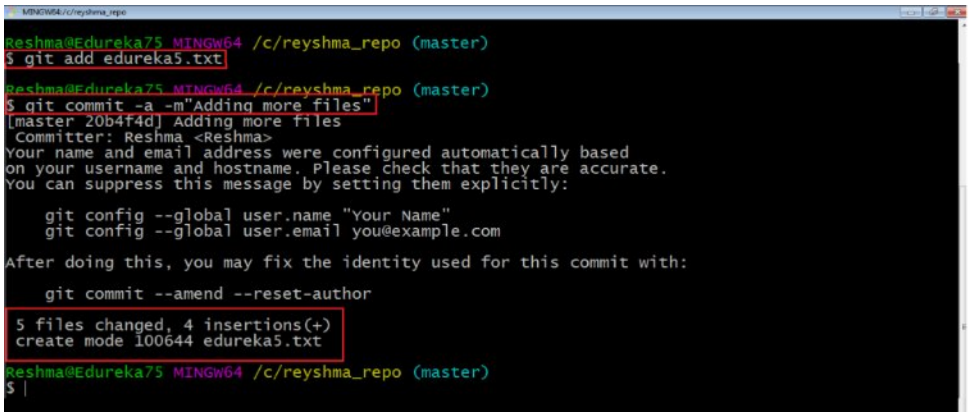
git commit -a

I have created two more text files in my working directory viz. *edureka5.txt* and *edureka6.txt* but they are not added to the index yet.

I am adding edureka5.txt using the command:

git add edureka5.txt

I have added *edureka5.txt* to the index explicitly but not *edureka6.txt* and made changes in the previous files. I want to commit all changes in the directory at once. Refer to the below snapshot.



```
MINGW64:/c/reyshma_repo
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git add edureka5.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git commit -a -m"Adding more files"
[master 20b4f4d] Adding more files
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 5 files changed, 4 insertions(+)
 create mode 100644 edureka5.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

This command will commit a snapshot of all changes in the working directory but only includes modifications to tracked files i.e. the files that have been added with git add at some point in their history. Hence, *edureka6.txt*was not committed because it was not added to the index yet. But changes in all previous files present in the repository were committed, i.e. *edureka1.txt*, *edureka2.txt*, *edureka3.txt*, *edureka4.txt* and *edureka5.txt*.
Now I have made my desired commits in my local repository.

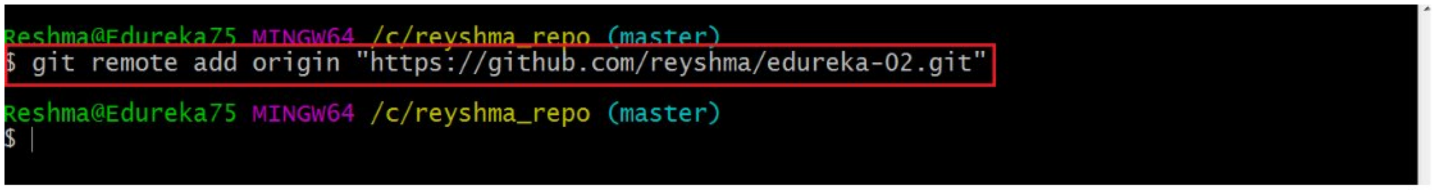
Note that before you affect changes to the central repository you should always pull changes from the central repository to your local repository to get updated with the work of all the collaborators that have been contributing in the central repository. For that we will use the pull command.

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.

But first, you need to set your central repository as origin using the command:

git remote add origin <link of your central repository>

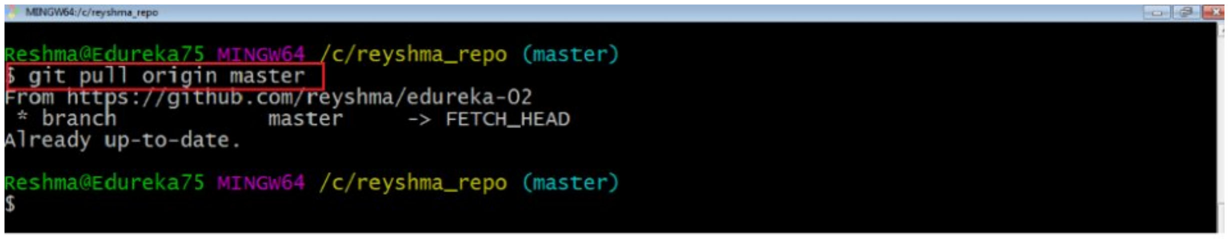


```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git remote add origin "https://github.com/reyshma/edureka-02.git"

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

Now that my origin is set, let us extract files from the origin using pull. For that use the command:

git pull origin master

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



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git pull origin master
From https://github.com/reyshma/edureka-02
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Since my local repository was already updated with files from master branch, hence the message is Already up-to-date. Refer to the screen shot above.

Note: One can also try pulling files from a different branch using the following command:

git pull origin <branch-name>

Your local Git repository is now updated with all the recent changes. It is time you make changes in the central repository by using the push command.

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 .

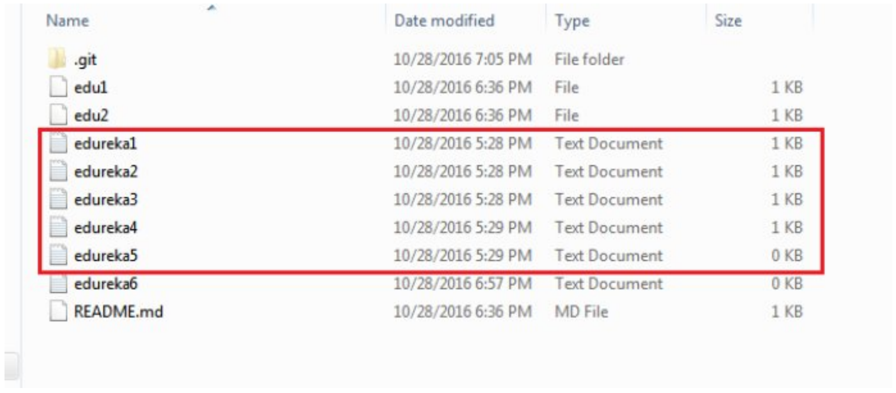
The use of git push is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

git push <remote>

Note *: This remote refers to the remote repository which had been set before using the pull command*.

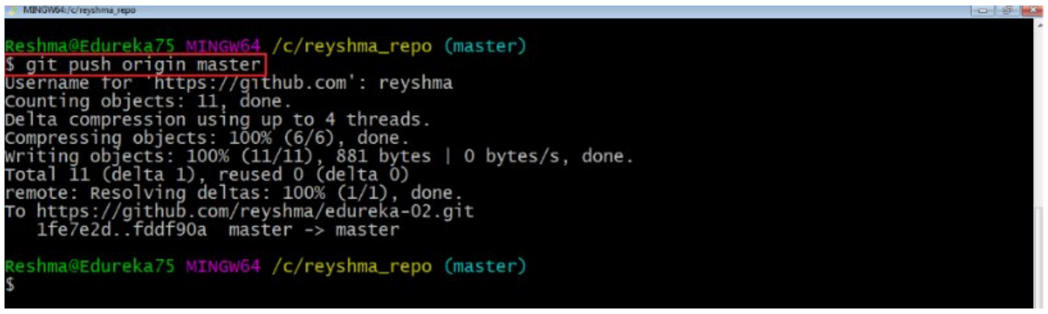
This pushes the changes from the local repository to the remote repository along with all the necessary commits and internal objects. This creates a local branch in the destination repository.

Let me demonstrate it for you.



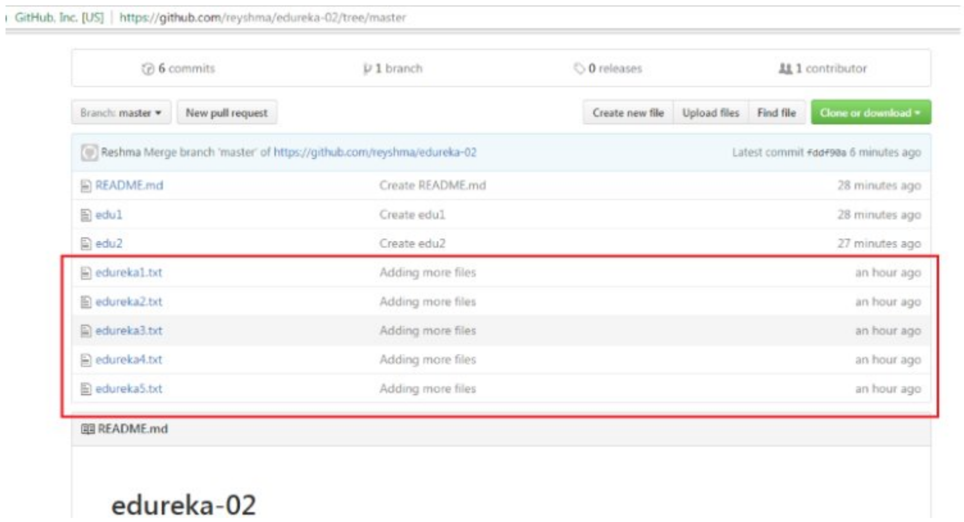
Name	Date modified	Type	Size
.git	10/28/2016 7:05 PM	File folder	
edu1	10/28/2016 6:36 PM	File	1 KB
edu2	10/28/2016 6:36 PM	File	1 KB
edureka1	10/28/2016 5:28 PM	Text Document	1 KB
edureka2	10/28/2016 5:28 PM	Text Document	1 KB
edureka3	10/28/2016 5:28 PM	Text Document	1 KB
edureka4	10/28/2016 5:29 PM	Text Document	1 KB
edureka5	10/28/2016 5:29 PM	Text Document	0 KB
edureka6	10/28/2016 6:57 PM	Text Document	0 KB
README.md	10/28/2016 6:36 PM	MD File	1 KB

The above files are the files which we have already committed previously in the commit section and they are all "*push-ready*". I will use the command git push origin master to reflect these files in the master branch of my central repository.



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git push origin master
Username for 'https://github.com': reyshma
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (11/11), 881 bytes | 0 bytes/s, done.
Total 11 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/reyshma/edureka-02.git
   1fe7e2d..fddf90a  master -> master

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Let us now check if the changes took place in my central repository.


GitHub, Inc. [US] | https://github.com/reyshma/edureka-02/tree/master

6 commits | 1 branch | 0 releases | 1 contributor

Reshma Merge branch 'master' of https://github.com/reyshma/edureka-02 — Latest commit fddf98a 6 minutes ago

README.md	Create README.md	28 minutes ago
edu1	Create edu1	28 minutes ago
edu2	Create edu2	27 minutes ago
edureka1.txt	Adding more files	an hour ago
edureka2.txt	Adding more files	an hour ago
edureka3.txt	Adding more files	an hour ago
edureka4.txt	Adding more files	an hour ago
edureka5.txt	Adding more files	an hour ago

README.md

edureka-02

Yes, it did. :-)

To prevent overwriting, Git does not allow push when it results in a non-fast forward merge in the destination repository.

Note: *A non-fast forward merge means an upstream merge i.e. merging with ancestor or parent branches from a child branch.*

To enable such merge, use the command below:

```
git push <remote> –force
```

The above command forces the push operation even if it results in a non-fast forward merge.

At this point of this Git Tutorial, I hope you have understood the basic commands of Git. Now, let’s take a step further to learn branching and merging in Git.

Branching

Branches in Git are nothing but pointers to a specific commit. Git generally prefers to keep its branches as lightweight as possible.

There are basically two types of branches viz. *local branches* and *remote tracking branches*.

A local branch is just another path of your working tree. On the other hand, remote tracking branches have special purposes. Some of them are:

- They link your work from the local repository to the work on central repository.
- They automatically detect which remote branches to get changes from, when you use git pull.

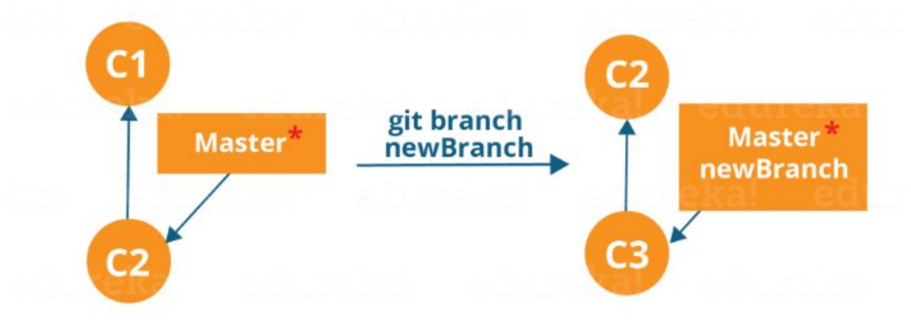
You can check what your current branch is by using the command:

```
git branch
```

The one mantra that you should always be chanting while branching is “branch early, and branch often”

To create a new branch we use the following command:

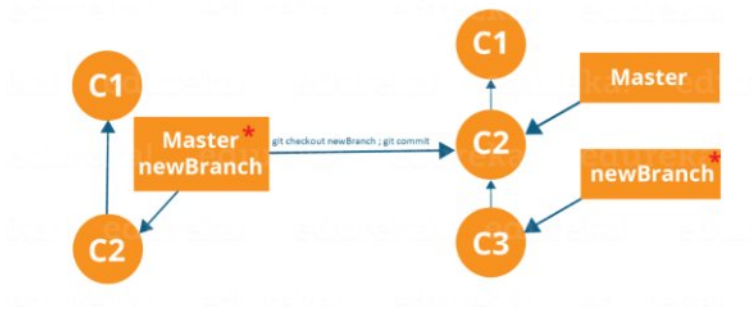
git branch <branch-name>



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.

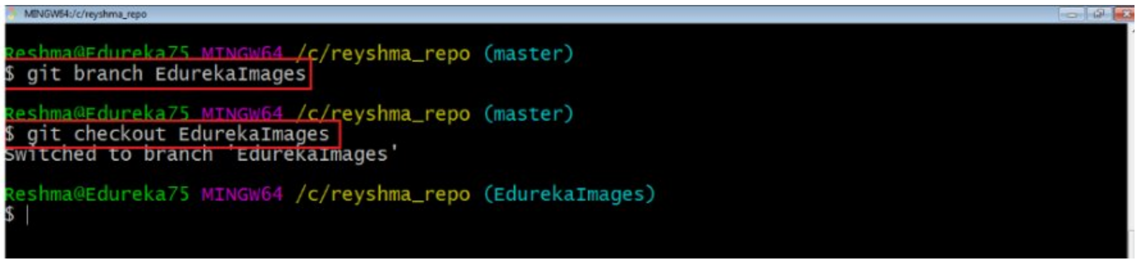


Branching includes the work of a particular commit along with all parent commits. As you can see in the diagram above, the newBranch has detached itself from the master and hence will create a different path.

Use the command below:

git checkout <branch_name> and then

git commit



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git branch EdurekaImages

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git checkout EdurekaImages
Switched to branch 'EdurekaImages'

Reshma@Edureka75 MINGW64 /c/reyshma_repo (EdurekaImages)
$ |
```

Here, I have created a new branch named “EdurekaImages” and switched on to the new branch using the command git checkout .

One shortcut to the above commands is:

```
git checkout -b[ branch_name]
```

This command will create a new branch and checkout the new branch at the same time.

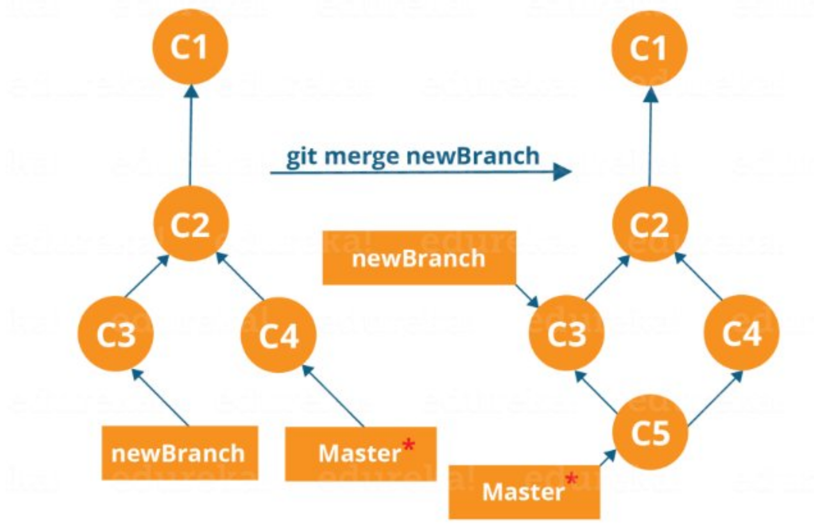
Now while we are in the branch EdurekaImages, add and commit the text file *edureka6.txt* using the following commands:

```
git add edureka6.txt
```

```
git commit -m”adding edureka6.txt”
```

Merging

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.



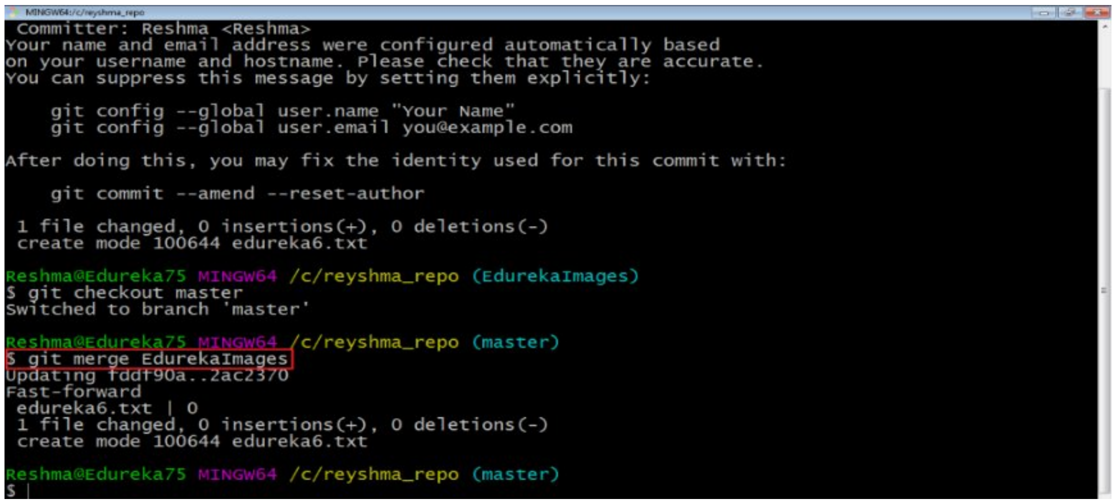
The diagram above shows us two different branches-> newBranch and master. Now, when we merge the work of newBranch into master, it creates a new commit which contains all the work of master and newBranch.

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.

Now, let us merge all of the work of the branch EdurekaImages into the master branch. For that I will first checkout the master branch with the command git checkout master and merge EdurekaImages with the command git merge EdurekaImages



```
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka6.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (EdurekaImages)
$ git checkout master
Switched to branch 'master'

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git merge EdurekaImages
Updating fddf90a..2ac2370
Fast-forward
 edureka6.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka6.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ 
```

As you can see above, all the data from the branch name are merged to the master branch. Now, the text file*edureka6.txt* has been added to the master branch.

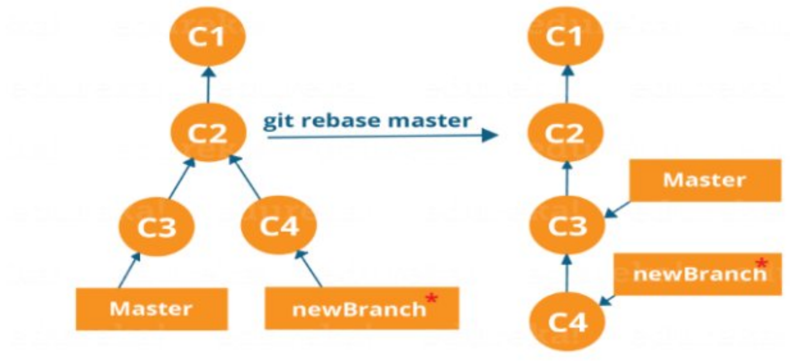
Merging in Git creates a special commit that has two unique parents.

Rebasing

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.

Let us see how it happens.

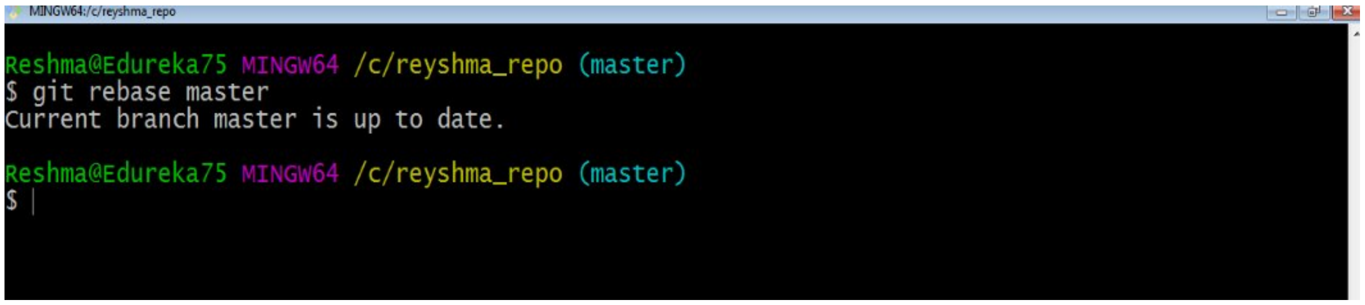


Now, our work from newBranch is placed right after master and we have a nice linear sequence of commits.

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



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git rebase master
Current branch master is up to date.

Reshma@Edureka75 MINGW64 /c/reyshma_repo (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.

Git Tutorial – Tips And Tricks

Now that you have gone through all the operations in this Git Tutorial, here are some tips and tricks you ought to know. :-)

- Archive your repository

Use the following command-

git archive master –format=zip –output= ../name-of-file.zip

It stores all files and data in a zip file rather than the .git directory.

Note that this creates only a single snapshot omitting version control completely. This comes in handy when you want to send the files to a client for review who doesn't have Git installed in their computer.

- Bundle your repository

It turns a repository into a single file.

Use the following command-

git bundle create ../repo.bundler master

This pushes the master branch to a remote branch, only contained in a file instead of a repository.

An alternate way to do it is:

cd..

git clone repo.bundle repo-copy -b master

cd repo-copy

git log

cd.. /my-git-repo

- Stash uncommitted changes

When we want to undo adding a feature or any kind of added data temporarily, we can “stash” them temporarily.

Use the command below:

```
git status
```

```
git stash
```

```
git status
```

And when you want to re apply the changes you “stash”ed ,use the command below:

```
git stash apply
```

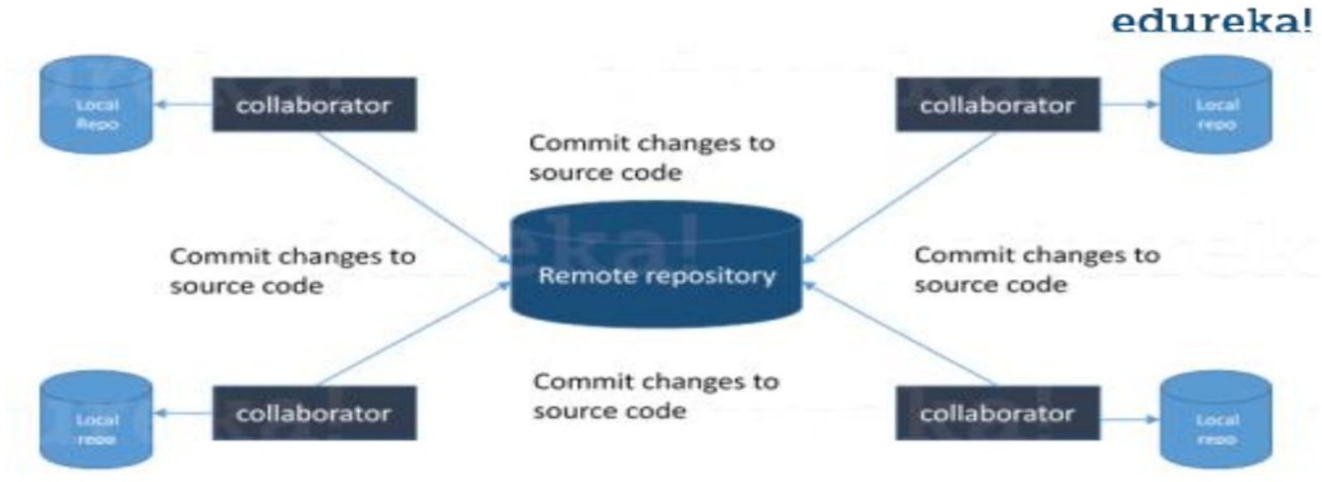
INTERVIEW QUS:

Q1. 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”.

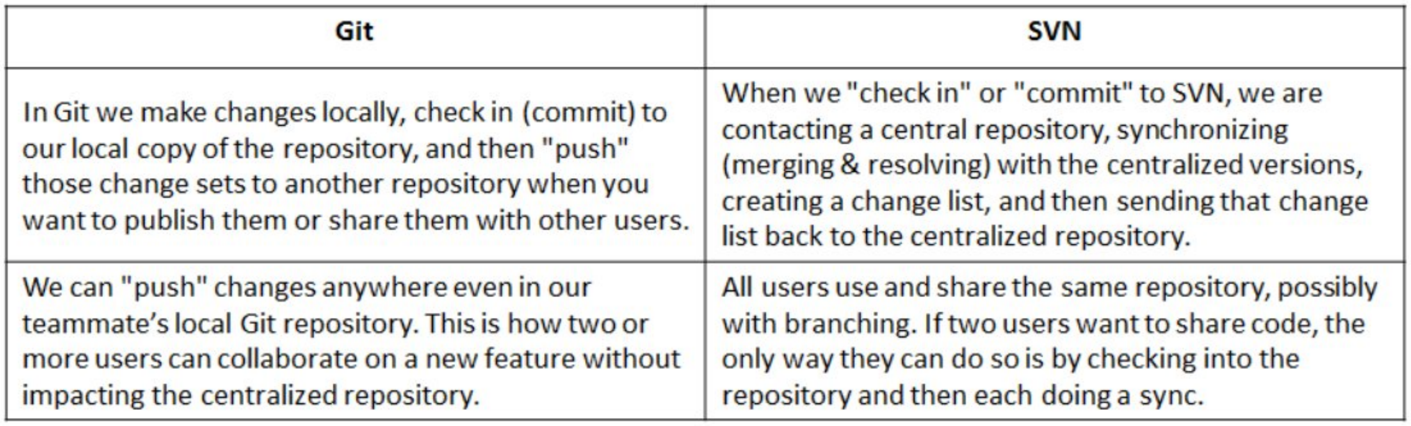

edureka!
Local Repo
collaborator
collaborator
Local repo
Commit changes to source code
Commit changes to source code
Remote repository
Commit changes to source code
Local repo
collaborator
Commit changes to source code
collaborator
Local repo

Now remember, you have mentioned SVN in the previous answer, so the next question in this Git Interview Questions blog will be related to the difference between Git and SVN

Q2. What is the difference between Git and SVN?

The proper answer for this according to me will be the architectural differences between Git and SVN. So the basic difference is that Git is distributed and SVN is centralized version control system.

Then explain the same by including the below mentioned differences:



Git	SVN
In Git we make changes locally, check in (commit) to our local copy of the repository, and then "push" those change sets to another repository when you want to publish them or share them with other users.	When we "check in" or "commit" to SVN, we are contacting a central repository, synchronizing (merging & resolving) with the centralized versions, creating a change list, and then sending that change list back to the centralized repository.
We can "push" changes anywhere even in our teammate's local Git repository. This is how two or more users can collaborate on a new feature without impacting the centralized repository.	All users use and share the same repository, possibly with branching. If two users want to share code, the only way they can do so is by checking into the repository and then each doing a sync.

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.

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.

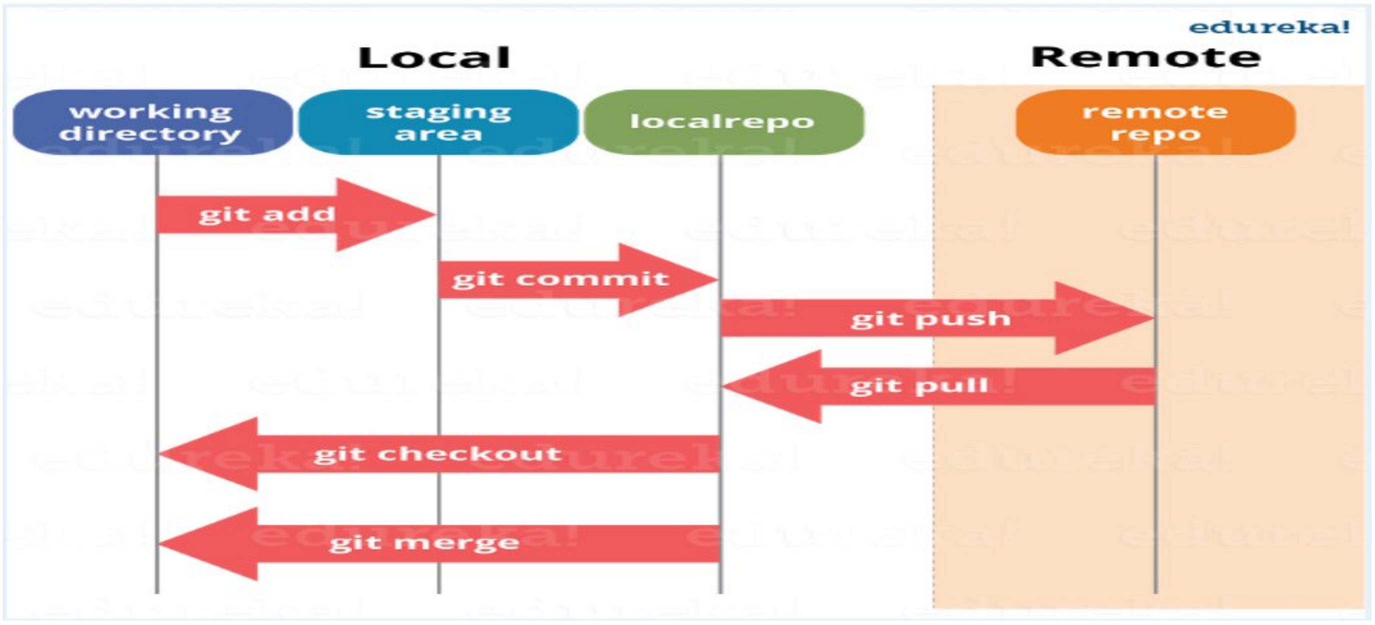
Git fetch is also used for the same purpose but it works in a slightly different way. When you perform a git fetch, it pulls all new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, git fetch must be followed 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 'git config'?

First tell why we need 'git config'.

Git uses your username to associate commits with an identity. The git config 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:

git config –global user.name "Your Name": This command will add username.
git config –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 "git init". 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
  git reset –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})"

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 test determines 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[-zfiles];then
exit0
fi
unfmtd=$(gofmt-l$files)
if[-zunfmtd];then
exit0
fi
echo“Some.gofilesarenotfmt’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:

- Featurebranching
  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.
- Taskbranching
  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.
- Releasebranching
  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 Atlassian Bitbucket Server.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.