Siva Git documentation

What is Git?

Git is currently the most popular implementation of a distributed version control system.

Git originates from the Linux kernel development and was founded in 2005 by Linus Torvalds. Nowadays it is used by many popular open source projects, e.g., the Android or the Eclipse developer teams, as well as many commercial organizations.

The core of Git was originally written in the programming language _C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python

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:

- Why Git came into existence?
- What is Git?
- Features of Git
- How Git plays a vital role in DevOps?
- How Microsoft and other companies are using Git

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:

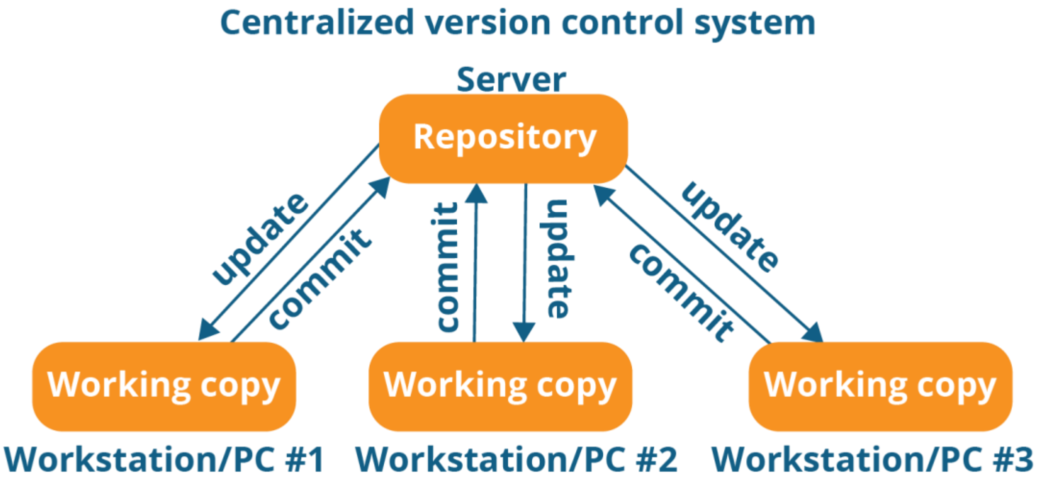
- Centralized Version Control System (CVCS)
- Distributed Version Control System (DVCS)

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:

Siva Git documentation



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.

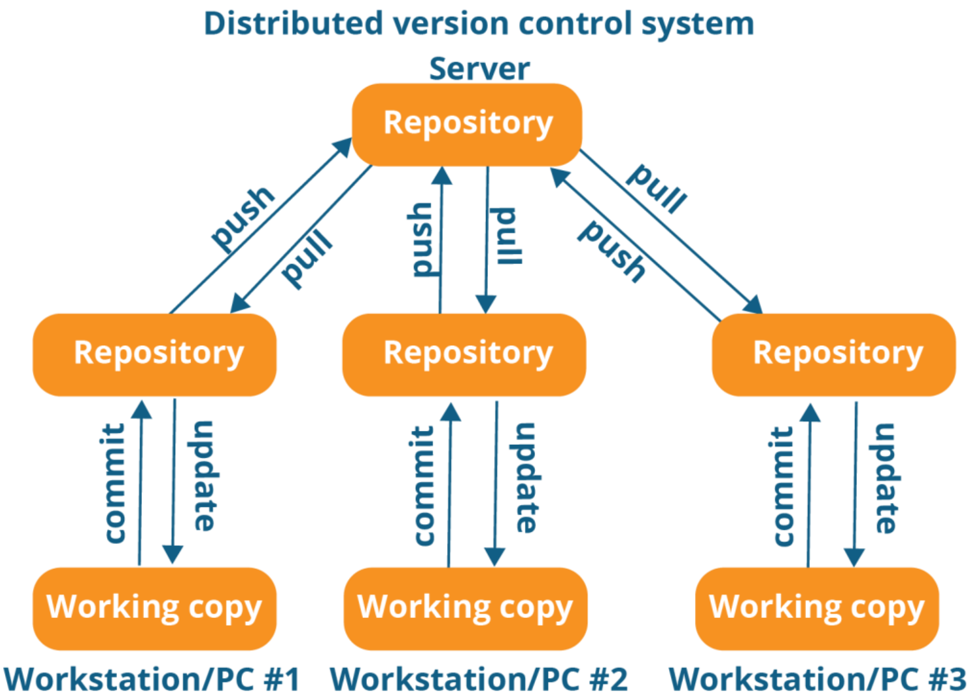
This is when Distributed VCS comes to the rescue.

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, it’s time we take a dive into what is Git.

What is Git – Features Of Git

Free and open source:

Siva Git documentation

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 compression technique.

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.

Siva Git documentation

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.

Siva Git documentation

Git repositories

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called _cloning. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories.

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server.

- bare repositories are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications.
- non-bare repositories target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation.

A *local non-bare Git repository* is typically called *local repository*

Working tree

A local repository provides at least one collection of files which originate from a certain version of the repository. This collection of files is called the *working tree*. It corresponds to a checkout of one version of the repository with potential changes done by the user.

The user can change the files in the *working tree* by modifying existing files and by creating and removing files. A file in the working tree of a Git repository can have different states. These states are the following:

- **Untracked:** the file is not tracked by the Git repository. This means that the file never staged nor committed.
- **tracked:** committed and not staged
- staged: staged to be included in the next commit
- **dirty / modified**: the file has changed but the change is not staged

After doing changes in the working tree, the user can add these changes to the Git repository or revert these changes.

Adding to a Git repository via staging and committing

After modifying your *working tree* you need to perform the following two steps to persist these changes in your local repository:

- add the selected changes to the *staging area* (also known as index) via the `git add` command
- commit the staged changes into the Git repository via the `git commit` command

This process is depicted in the following graphic.

The git add command stores a snapshot of the specified files in the staging area. It allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

Some tools and Git user prefer the usage of the *index* instead of staging area. Both terms mean the same thing.

After adding the selected files to the staging area, you can *commit* these files to add them permanently to the Git repository. _ Committing_ creates a new persistent snapshot (called *commit* or *commit object*) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The *staging area* keeps track of the snapshots of the files until the staged changes are committed.

For committing the staged changes you use the git commit command.

If you commit changes to your Git repository, you create a new *commit object* in the Git repository. See Commit object (commit) for information about the commit object.

Synchronizing with other Git repositories (remote repositories)

Git allows the user to synchronize the local repository with other (remote) repositories.

Users with sufficient authorization can send new version in their local repository to to remote repositories via the *push* operation. They can also integrate changes from other repositories into their local repository via the *fetch* and *pull* operation.

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.

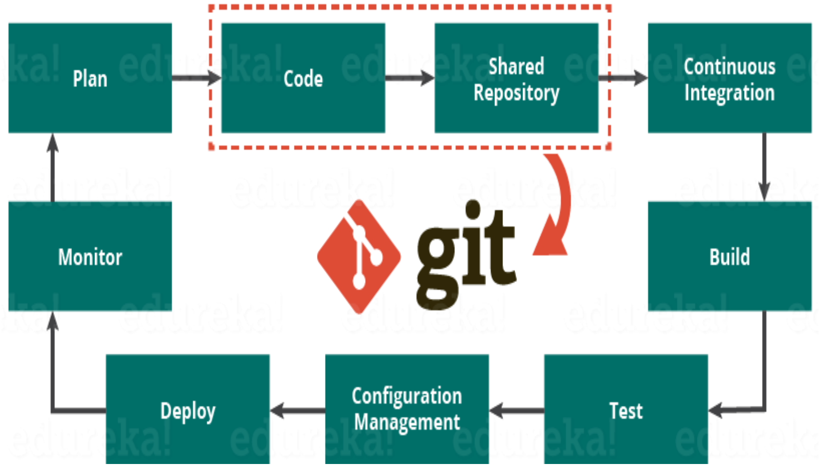
What is Git – Role of Git in DevOps?

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.

Siva Git documentation

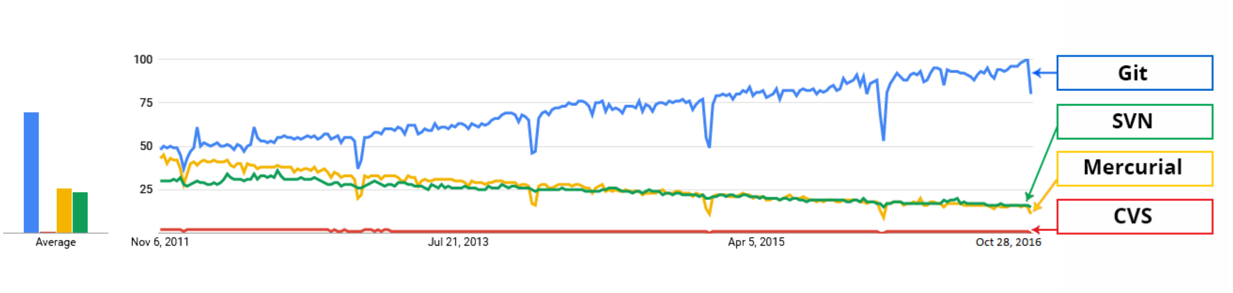


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration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

Siva Git documentation

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.

At this point you know what Git is; now, let us learn to use commands and perform operations in my next ***Git Tutorial*** blog.

Git Tutorial

I hope that you have gone through the basic concepts and terminologies of Git and learned all about Version Control in my first blog of the Git Tutorial series. If you haven't, please check out my ***previous blog*** to get a better understanding of Git.

In this Git Tutorial, you will learn:

- Commands in Git
- Git operations
- And some tips and tricks to manage your project effectively with Git.

Now that you know what this Git Tutorial will bring to you, let us begin :-)

Before starting with the commands and operations let us first understand the primary motive of Git.

The motive of Git is to manage a project or a set of files as they change over time. Git stores this information in a data structure called a Git repository. The repository is the core of Git.

To be very clear, a Git repository is the directory where all of your project files and the related metadata resides.

Git records the current state of the project by creating a tree graph from the index. It is usually in the form of a Directed Acyclic Graph (DAG).

Before you go ahead, check out this video on GIT tutorial to have better in-sight.

Git Tutorial – Operations & Commands

Some of the basic operations in Git are:

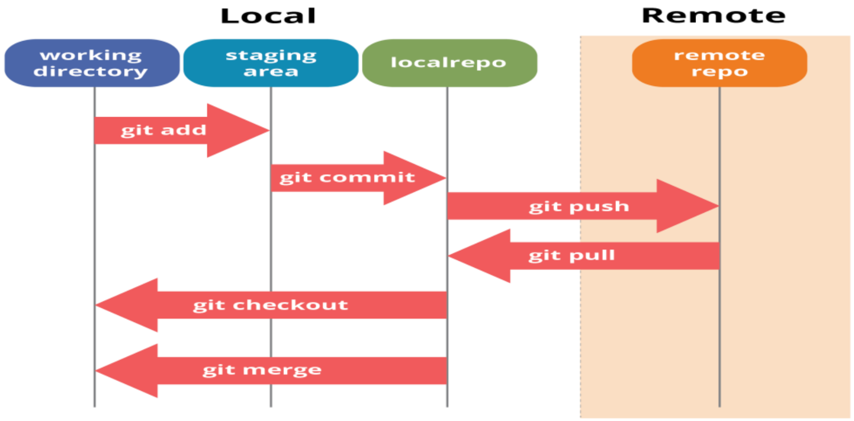
1. Initialize
2. Add
3. Commit
4. Pull
5. Push

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:

Siva Git documentation

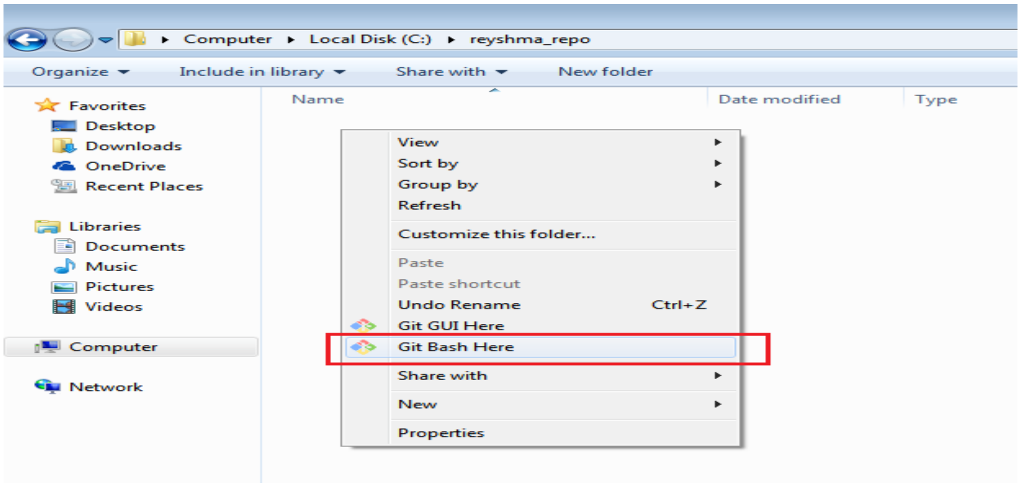


If you understand the above diagram well and good, but if you don’t, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

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**.

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.

Siva Git documentation

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 *GoldenTech1.txt* and *GoldenTech2.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:

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 **git add** command 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.

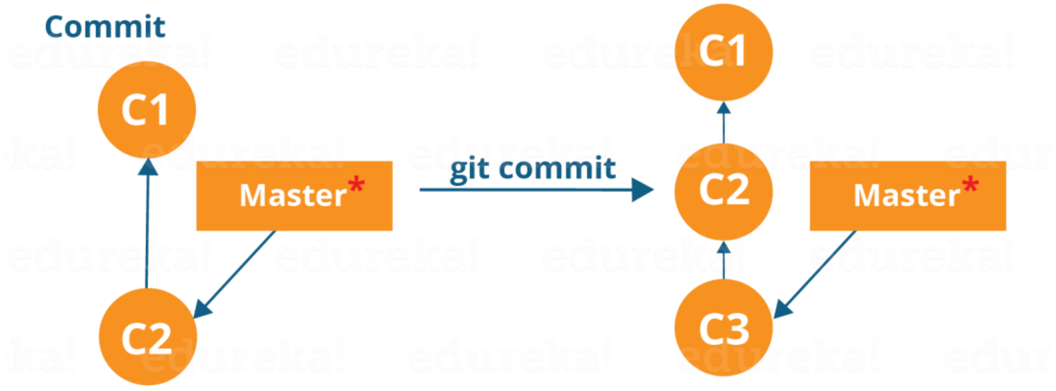
I have created two more files *GoldenTech3.txt* and *GoldenTech.txt*. Let us add the files using the command **git add -A**. This command will add all the files to the index which are in the directory but not updated in the index yet.

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 diagram below:

Siva Git documentation



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.

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 GoldenTech5.txt using the command:

git add GoldenTech5**.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.

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, *GoldenTech6.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. *GoldenTech1.txt*, *GoldenTech2.txt*, *GoldenTech3.txt*, *GoldenTech4.tx*and *GoldenTech5.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>

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.

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.

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.

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

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**.

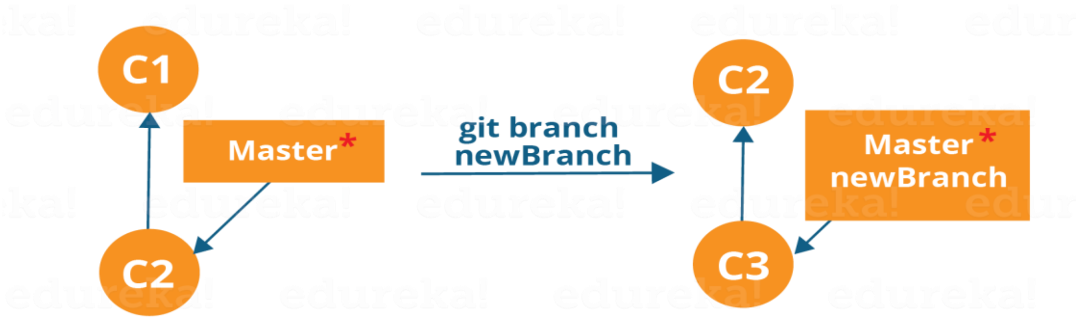
You can check what your current branch is by using the command:

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:

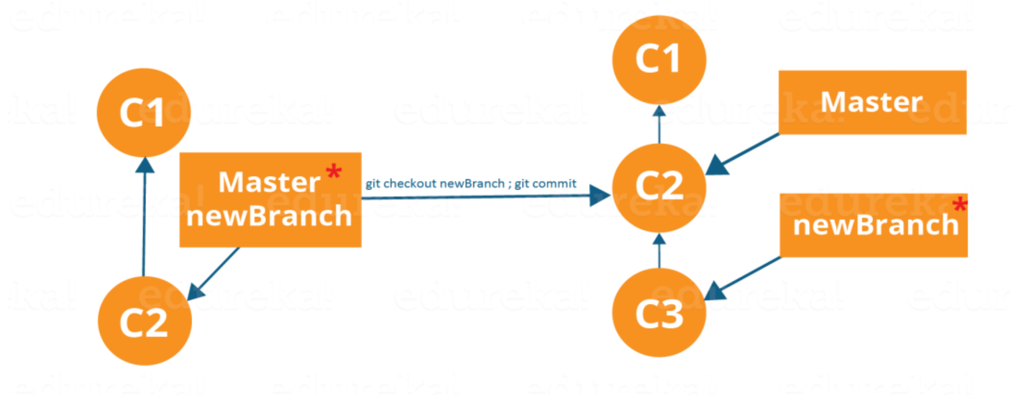
git branch <branch-name>



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 new Branch 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

Here, I have created a new branch named" 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.

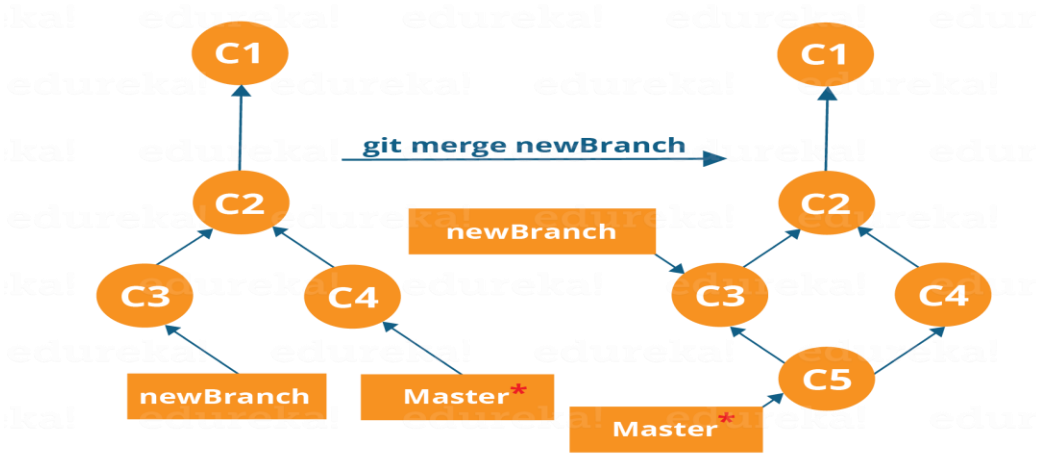
Now while we are in the branch add and commit the text file *edureka6.txt* using the following commands:

git add GoldenTech6.txt

git commit -m"adding GoldenTech.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-> new Branch and master. Now, when we merge the work of new Branch into master, it creates a new commit which contains all the work of master and new Branch.

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

git merge <branch_name>

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

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

As you can see above, all the data from the branch name are merged to the master branch. Now, the text GoldenTechnology*6.txt* has been added to the master branch.

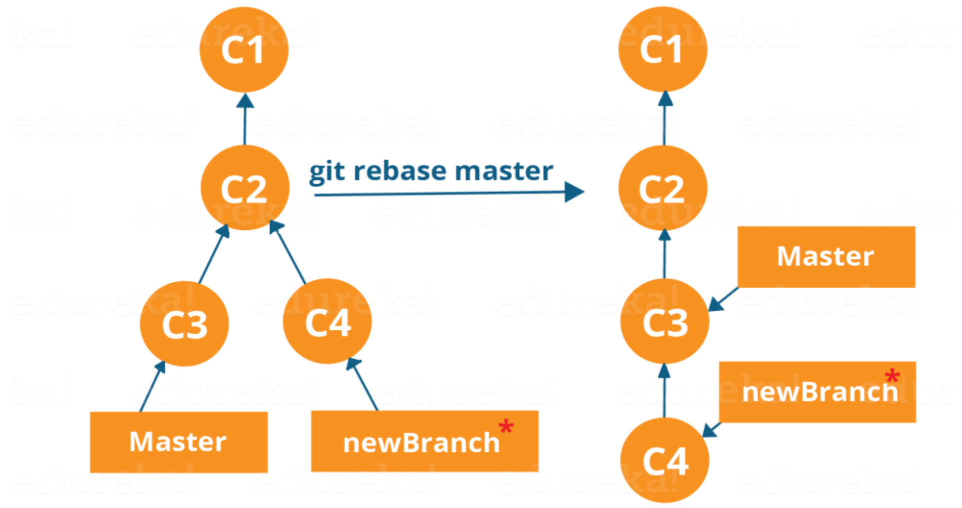
Merging in Git creates a special commit that has two unique parents.

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 new Branch is placed right after master and we have a nice linear sequence of commits.

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

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

```
git rebase master
```

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

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
```

I hope you have enjoyed this Git Tutorial and learned the commands and operations in Git. Let me know if you want to know more about Git in the comments section below :-

Install Git

Let me guide you through the process to install Git in your system through this blog. In case you want to know more about Git don't forget to checkout **this blog**.

In this Install Git blog you will learn:

- How to install Git in Windows
- How to install Git in Ubuntoo
- How to make your repositories on GitHub

Before you move ahead, check out this video on GIT installation.

So, without any further ado, let us begin to install Git on a Windows system first.

Install Git On Windows

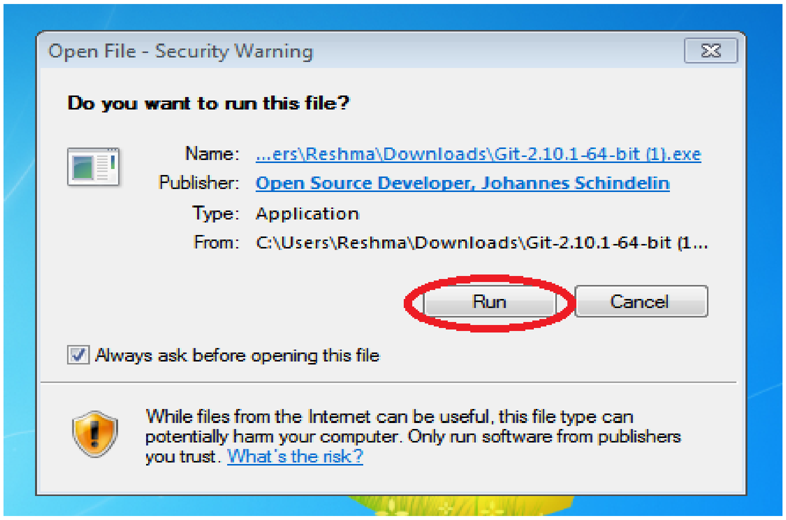
Step 1: To download the latest version of Git, click on the link below:

Download Git for Windows

Great! Your file is being downloaded.

Step 2:

After your download is completed, run the .exe file in your system.



Open File - Security Warning

Do you want to run this file?

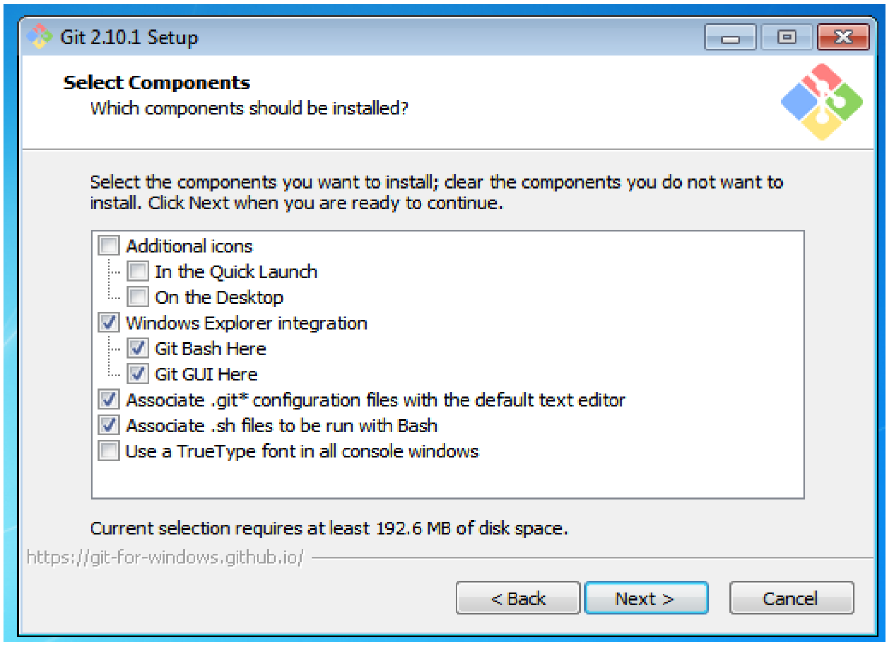
Name: ...ers\Reshma\Downloads\Git-2.10.1-64-bit (1).exe
Publisher: Open Source Developer, Johannes Schindelin
Type: Application
From: C:\Users\Reshma\Downloads\Git-2.10.1-64-bit (1...

☑ Always ask before opening this file

While files from the Internet can be useful, this file type can potentially harm your computer. Only run software from publishers you trust. What's the risk?

Step 3:

After you have pressed the run button and agreed to the license, you will find a window prompt to select components to be installed.



Git 2.10.1 Setup

Select Components

Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- ☐ Additional icons
  - ☐ In the Quick Launch
  - ☐ On the Desktop
- ☑ Windows Explorer integration
  - ☑ Git Bash Here
  - ☑ Git GUI Here
- ☑ Associate .git* configuration files with the default text editor
- ☑ Associate .sh files to be run with Bash
- ☐ Use a TrueType font in all console windows

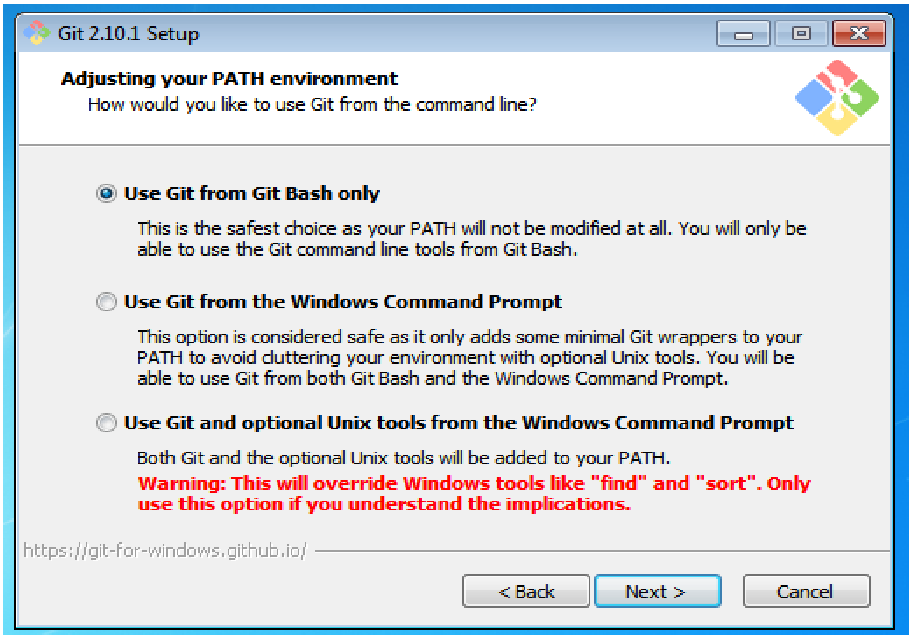
Current selection requires at least 192.6 MB of disk space.

https://git-for-windows.github.io/

After you have made selection of your desired components, Click on ‘Next>’.

Step 4:

This will take you to the next prompt window to make choices for adjusting your path environment. This is where you decide how do you want to use Git.



Git 2.10.1 Setup

Adjusting your PATH environment

How would you like to use Git from the command line?

● **Use Git from Git Bash only**

This is the safest choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

○ **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from both Git Bash and the Windows Command Prompt.

○ **Use Git and optional Unix tools from the Windows Command Prompt**

Both Git and the optional Unix tools will be added to your PATH.

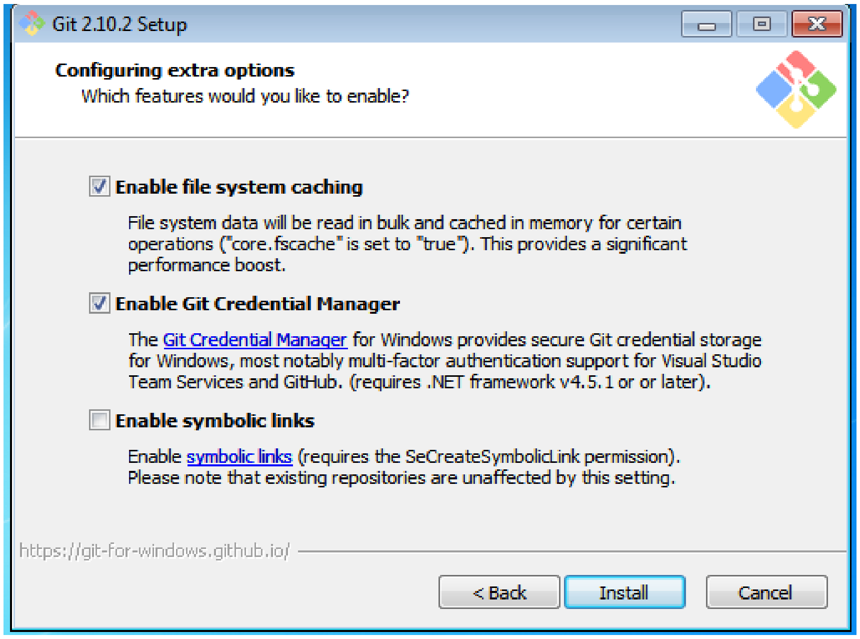
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

https://git-for-windows.github.io/

You can select any of the three options according to your needs. But for rookies I recommend using "Use Git From Git Bash Only"

Step 5:

The next step is to choose features for your Git. You get three options and you can choose any of them, all of them or none of them as per your needs. Let me tell you what these features are:



Git 2.10.2 Setup

Configuring extra options

Which features would you like to enable?

☑ **Enable file system caching**

File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

☑ **Enable Git Credential Manager**

The Git Credential Manager for Windows provides secure Git credential storage for Windows, most notably multi-factor authentication support for Visual Studio Team Services and GitHub. (requires .NET framework v4.5.1 or or later).

☐ **Enable symbolic links**

Enable symbolic links (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

https://git-for-windows.github.io/

The first is the option to enable file system caching.

Caching is enabled through Cache manager, which operates continuously while Windows is running. File data in the system file cache is written to the disk at intervals determined by the operating system, and the memory previously used by that file data is freed.

The second option is to enable Git Credential Manager.

The Git Credential Manager for Windows (GCM) is a credential helper for Git. It securely stores your credentials in the Windows CM so that you only need to enter them once for each remote repository you access. All future Git commands will reuse the existing credentials.

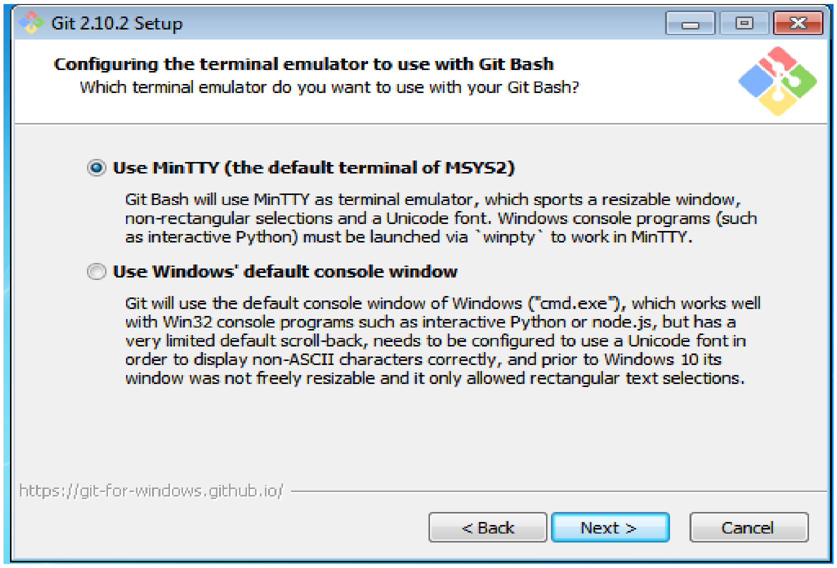
The third option is to enable symbolic links.

Symbolic links or symlinks are actually advanced shortcuts. You can create symbolic links for each individual file or folder, and these will appear like they are stored in the folder with symbolic link.

I have selected the first two features only.

Step 6:

Choose your terminal.



Configuring the terminal emulator to use with Git Bash
Which terminal emulator do you want to use with your Git Bash?

● **Use MinTTY (the default terminal of MSYS2)**

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via `winpty` to work in MinTTY.

○ **Use Windows' default console window**

Git will use the default console window of Windows ("cmd.exe"), which works well with Win32 console programs such as interactive Python or node.js, but has a very limited default scroll-back, needs to be configured to use a Unicode font in order to display non-ASCII characters correctly, and prior to Windows 10 its window was not freely resizable and it only allowed rectangular text selections.

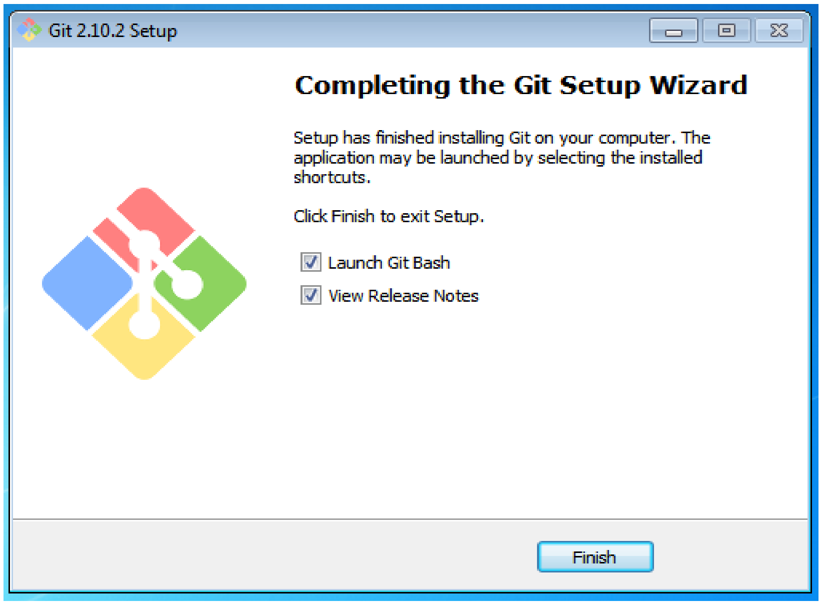
You can choose one from the options.

The default terminal of MYSYS2 which is a collection of GNU utilities like bash, make, gawk and grep to allow building of applications and programs which depend on traditionally UNIX tools to be present.

Or you can choose the window’s default console window (cmd.exe).

Step 7:

Now you have got all you need. Select “Launch Git Bash” and click on “Finish”.



Git 2.10.2 Setup

Completing the Git Setup Wizard

Setup has finished installing Git on your computer. The application may be launched by selecting the installed shortcuts.

Click Finish to exit Setup.

☑ Launch Git Bash

☑ View Release Notes

This will launch Git Bash on your screen which looks like the snapshot below:

Git important commands

For example, you can ask Git to show all commits which happened between HEAD and HEAD~4.

```
git log HEAD~4..HEAD
```

This also works for branches. To list all commits which are in the "master" branch but not in the "testing" branch, use the following command.

```
git log testing..master
```

You can also list all commits which are in the "testing" but not in the "master" branch.

```
git log master..testing
```

Commit ranges with the triple dot operator

The triple dot operator allows you to select all commits which are reachable either from commit c1 or commit c2 but not from both of them.

This is useful to show all commits in two branches which have not yet been combined.

Siva Git documentation

```
# show all commits which
# can be reached by master or testing
# but not both
git log master...testing git help [command to get help for]
```

See all possible commands, use the git help --all command.

Git supports for several commands a short and a long version, similar to other Unix commands. The short version uses a single hyphen and the long version uses two hyphen. The following two commands are equivalent.

```
git commit -m "This is a message"

git commit --message "This is a message"
```

Installation of the Git command line tooling

Ubuntu, Debian and derived systems

On Ubuntu and similar systems you can install the Git command line tool via the following command:

```
sudo apt-get install git
```

Fedora, Red Hat and derived systems

On Fedora, Red Hat and similar systems you can install the Git command line tool via the following command:

```
dnf install git
```

Other Linux systems

To install Git on other Linux distributions please check the documentation of your distribution. The following listing contains the commands for the most popular ones.

```
# Arch Linux
sudo pacman -S git

# Gentoo
sudo emerge -av git

# SUSE
sudo zypper install git
```

Windows

A Windows version of Git can be found on the Git download page. This website provides native installers for each operating system. The homepage of the Windows Git project is git for window.

Siva Git documentation

User credential configuration

You have to configure at least your user and email address to be able to commit to a Git repository because this information is stored in each commit.

```
# configure the user which will be used by Git
# this should be not an acronym but your full name
git config --global user.name "Firstname Lastname"

# configure the email address
git config --global user.email "your.email@example.org"
```

Push configuration

If your are using Git in a version below 2.0 you should also execute the following command.

```
# set default so that only the current branch is pushed
git config --global push.default simple
```

This configures Git so that the `git push` command pushes only the active branch to your Git remote repository. As of Git version 2.0 this is the default and therefore it is good practice to configure this behavior.

Mac OS

The easiest way to install Git on a Mac is via the Git download page and to download and run the installer for Mac OS X.

Git is also installed by default with the Apple Developer Tools on Mac OS X

Color Highlighting

The following commands enables color highlighting for Git in the console.

```
git config --global color.ui auto
```

See the current status of your repository

The `git status` command shows the status of the working tree, i.e. which files have changed, which are staged and which are not part of the staging area. It also shows which files have conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

Run it via the following command.

```
git status
```

The output looks similar to the following listing.

```
On branch master

Initial commit

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

    datafiles/
    test01
    test02
    test03

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

Add changes to the staging area

Before committing changes to a Git repository, you need to mark the changes that should be committed with the `git add` command. This command allows adding changes in the file system to the staging area. It creates a snapshot of the affected files. You can add all changes to the staging area with the `.` option or changes in individual files but specifying a file pattern as option.

```
# add all files to the index of the Git repository
git add .
```

Afterwards run the `git status` command again to see the current status. The following listing shows the output of this command.

```
On branch master

Initial commit

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

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03
```

Change files that are staged

In case you change one of the staged files before committing, you need to add the changes again to the staging area, to commit the new changes. This is because Git creates a snapshot of the content of a staged file. All new changes must again be staged.

```
# append a string to the test03 file
echo "foo2" >> test03
```

```
# see the result
git status
```

Validate that the new changes are not yet staged.

```
On branch master

Initial commit

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

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03

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:   test03
```

Add the new changes to the staging area.

```
# add all files to the index of the Git repository
git add .
```

Use the `git status` command again to see that all changes are staged.

```
On branch master

Initial commit

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

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03
```

Commit staged changes to the repository

After adding the files to the Git staging area, you can commit them to the Git repository with the `git commit` command. This creates a new commit object with the staged changes in the Git repository and the HEAD reference points to the new commit. The `-m` parameter (or its long version: `--message`) allows you to specify the commit message. If you leave this parameter out, your default editor is started and you can enter the message in the editor.

```
# commit your file to the local repository
git commit -m "Initial commit"
```

> Git also offers a mode that lets you choose interactively which changes you want to commit. After you quit th
> mode you will be asked to provide a commit message in your $EDITOR.
>
> ```
> git commit --interactive
> ```

Viewing the Git commit history

The Git operations you performed have created a local Git repository in the `.git` folder and added all files to this repository via one commit. Run the `git log` command to see the history.

```
# show the Git log for the change
git log
```

You see an output similar to the following.

```
commit 30605803fcbd507df36a3108945e02908c823828
Author: Lars Vogel <Lars.Vogel@vogella.com>
Date:   Mon Dec 1 10:43:42 2014 +0100

    Initial commit
```

Viewing the changes of a commit

Use the `git show` command to see the changes of a commit. If you specify a commit reference as third parameter, this is used to determine the changes, otherwise the *HEAD* reference is used.

Review the resulting directory structure

Review the resulting directory structure. Your directory contains the Git repository as well as the Git working tree for your files. This directory structure is depicted in the following screenshot

Remove files

If you delete a file, you use the `git add .` command to add the deletion of a file to the staging area.

```
# remove the "test03" file
rm test03
# add and commit the removal
git add .
# if you use Git version < 2.0 use: git add -A .
git commit -m "Removes the test03 file"
```

Alternatively you can use the `git rm` command to delete the file from your working tree and record the deletion of the file in the staging area.

Revert changes in files in the working tree

Use the git checkout command to reset a tracked file (a file that was once staged or committed) to its latest staged or commit state. The command removes the changes of the file in the working tree. This command cannot be applied to files which are not yet staged or committed.

```
echo "useless data" >> test02
echo "another unwanted file" >> unwantedfile.txt

# see the status
git status

# remove unwanted changes from the working tree
# CAREFUL this deletes the local changes in the tracked file
git checkout test02

# unwantedstaged.txt is not tracked by Git simply delete it
rm unwantedfile.txt
```

If you use git status command to see that there are no changes left in the working directory.

```
On branch master
nothing to commit, working directory clean
```

> Use this command carefully. The git checkout command deletes the unstaged and uncommitted changes of tracked files in the working tree and it is not possible to restore this deletion via Git.

Correct the changes of the commit with git amend

The git commit --amend command makes it possible to rework the changes of the last commit. It creates a new commit with the adjusted changes.

> The amended commit is still available until a clean-up job removes it. But it is not included in the git log outp hence it does not distract the user. See git reflog for details.

Assume the last commit message was incorrect as it contained a typo. The following command corrects this via the --amend parameter.

```
# assuming you have something to commit
git commit -m "message with a tpyo here"
# amend the last commit
git commit --amend -m "More changes - now correct"
```

You should use the git --amend command only for commits which have not been pushed to a public branch of another Git repository. The git --amend command creates a new commit ID and people may have based their work already on the existing commit. If that would be the case, they would need to migrate their work based on the new commit.

You can create a new branch via the `git branch [newname]` command. This command allows to specify the commit (commit id, tag, remote or local branch) to which the branch pointer original points. If not specified, the commit to which the HEAD reference points is used to create the new branch.

```
# syntax: git branch <name> <hash>
# <hash> in the above is optional
git branch testing
```

Checkout branch

To start working in a branch you have to *checkout* the branch. If you *checkout* a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

The following commands demonstrate how you switch to the branch called *testing*, perform some changes in this branch and switch back to the branch called *master*.

```
# switch to your new branch
git checkout testing

# do some changes
echo "Cool new feature in this branch" > test01
git commit -a -m "new feature"

# switch to the master branch
git checkout master

# check that the content of
# the test01 file is the old one
cat test01
```

To create a branch and to switch to it at the same time you can use the `git checkout` command with the `-b` parameter.

```
# create branch and switch to it
git checkout -b bugreport12

# creates a new branch based on the master branch
# without the last commit
git checkout -b mybranch master~1
```

Rename a branch

Renaming a branch can be done with the following command.

```
# rename branch
git branch -m [old_name] [new_name]
```

Delete a branch

To delete a branch which is not needed anymore, you can use the following command. You may get an error message that there are uncommited changes if you did the previous examples step by step. Use force delete (uppercase -D) to delete it anyway.

```
# delete branch testing
git branch -d testing
# force delete testing
git branch -D testing
# check if branch has been deleted
git branch
```

Push changes of a branch to a remote repository

You can push the changes in a branch to a remote repository by specifying the target branch. This creates the target branch in the remote repository if it does not yet exist.

If you do not specify the remote repository, the origin is used as default

```
# push current branch to a branch called "testing" to remote repository
git push origin testing

# switch to the testing branch
git checkout testing

# some changes
echo "News for you" > test01
git commit -a -m "new feature in branch"

# push current HEAD to origin
git push

# make new branch
git branch anewbranch
# some changes
echo "More news for you" >> test01
git commit -a -m "a new commit in a feature branch"
# push anewbranch to the master in the origin
git push origin anewbranch:master

# get the changes into your local master
git checkout master
git pull
```

This way you can decide which branches you want to push to other repositories and which should be local branches. You learn more about branches and remote repositories in Remote tracking branches.

Siva Git documentation

Switching branches with untracked files

Untracked files (never added to the staging area) are unrelated to any branch. They exist only in the working tree and are ignored by Git until they are committed to the Git repository. This allows you to create a branch for upstaged and uncommitted changes at any point in time.

Switching branches with uncommitted changes

Similar to untracked files you can switch branches with unstaged or staged modifications which are not yet committed.

You can switch branches if the modifications do not conflict with the files from the branch.

If Git needs to modify a changed file during the checkout of a branch, the checkout fails with a "checkout conflict" error. This avoids that you lose changes in your files.

In this case the changes must be committed, reverted or stashed (see The git stash command). You can also always create a new branch based on the current HEAD.

Differences between branches

To see the difference between two branches you can use the following command.

```
# shows the differences between
# current head of master and your_branch

git diff master your_branch
```

You can use commit ranges as described in Commit ranges with the double dot operator and Commit ranges with the triple dot operator. For example, if you compare a branch called *your_branch* with the *master* branch the following command shows the changes in *your_branch* and *master* since these branches diverged.

```
# shows the differences in your
# branch based on the common
# ancestor for both branches

git diff master...your_branch
```

See Viewing changes with git diff and git show for more examples of the `git diff` command

What are tags?

Git has the option to *tag* a commit in the repository history so that you find it easier at a later point in time. Most commonly, this is used to tag a certain version which has been released.

If you tag a commit, you create an annotated or lightweight tag.

Siva Git documentation

Lightweight and annotated tags

Git supports two different types of tags, lightweight and annotated tags.

A *lightweight tag* is a pointer to a commit, without any additional information about the tag. An *annotated tag* contains additional information about the tag, e.g., the name and email of the person who created the tag, a tagging message and the date of the tagging. Annotated tags can also be signed and verified with *GNU Privacy Guard (GPG).*

Naming conventions for tags

Tags are frequently used to tag the state of a release of the Git repository. In this case they are typically called *release tags*.

Convention is that release tags are labeled based on the [major].[minor].[patch] naming scheme, for example "1.0.0". Several projects also use the "v" prefix.

The idea is that the *patch* version is incremented if (only) backwards compatible bug fixes are introduced, the *minor* version is incremented if new, backwards compatible functionality is introduced to the public API and the *major* version is incremented if any backwards incompatible changes are introduced to the public API.

For the detailed discussion on naming conventions please see the following URL: Semantic versioning.

List tags

You can list the available tags via the following command:

```
git tag
```

Search by pattern for a tag

You can use the `-l` parameter in the `git tag` command to search for a pattern in the tag.

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

Creating lightweight tags

To create a lightweight tag don’t use the `-m`, `-a` or `-s` option.

The term *build* describes the conversion of your source code into another state, e.g., converting Java sources to an executable `JAR` file. Lightweight tags in Git are often used to identify the input for a build. Frequently this does not require additional information other than a build identifier or the timestamp.

```
# create lightweight tag
git tag 1.7.1

# see the tag
```

```
git show 1.7.1
```

Creating annotated tags

You can create a new annotated tag via the `git tag -a` command. An annotated tag can also be created using the `-m` parameter, which is used to specify the description of the tag. The following command tags the current active HEAD.

```
# create tag
git tag 1.6.1 -m 'Release 1.6.1'

# show the tag
git show 1.6.1
```

You can also create tags for a certain commit id.

```
git tag 1.5.1 -m 'version 1.5' [commit id]
```

Creating signed tags

You can use the option `-s` to create a signed tag. These tags are signed with *GNU Privacy Guard (GPG)* and can also be verified with GPG. For details on this please see the following URL: Git tag manpage.

Checkout tags

If you want to use the code associated with the tag, use:

```
git checkout <tag_name>
```

> If you checkout a tag, you are in the *detached head mode* and commits created in this mode are harder to find after you checkout a branch again. See Detached HEAD for details.

Push tags

By default the `git push` command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.

```
# push a tag or branch called tagname
git push origin [tagname]

# to explicitly push a tag and not a branch
git push origin tag <tagname>

# push all tags
git push –tags
```

Siva Git documentation

1. **What are branches**
2. **How to create a branch**
3. **How to checkout a branch**
4. **How to merge branch to master**
5. **How to delete a branch (local and remote)**

Step 1. Create Branch

```
git branch <branch name>   or git branch –b <branch name>
```

step 2. Checkout branch

```
git checkout <branch name>
```

step 3. Merge new branch in master branch

```
git merge <new branch name>
```

step 4. Deleting the branch

```
git branch –d <branch name>    ---  This will only remove from local
```

```
git push origin –delete <branch name>  ---  This will remove from remote server repository.
```