**Notes for MCA-II (Semester- IV)**

**Subject :- DevOps**

**(Subject Code:- IT-41**

**Chapter: 2] Version Control – GIT**

**2.1 Introduction to GIT :-**

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is a free software distributed under the terms of the GNU General Public License version 2

Git is a modern and widely used **distributed version control** system in the world. It is developed to manage projects with high speed and efficiency. The version control system allows us to monitor and work together with our team members at the same workspace.

**2.2 What is GIT ?**

**Git** is an **open-source distributed version control system**. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

* Tracking code changes
* Tracking who made changes
* Coding collaboration

**Working with Git**

* Initialize Git on a folder, making it a **Repository**
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is changed, added or deleted, it is considered **modified**
* You select the modified files you want to **Stage**
* The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
* Git allows you to see the full history of every commit.
* You can revert back to any previous commit.
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

**Why Git?**

* Over 70% of developers use Git!
* Developers can work together from anywhere in the world.
* Developers can see the full history of the project.
* Developers can revert to earlier versions of a project.

**Features of Git**

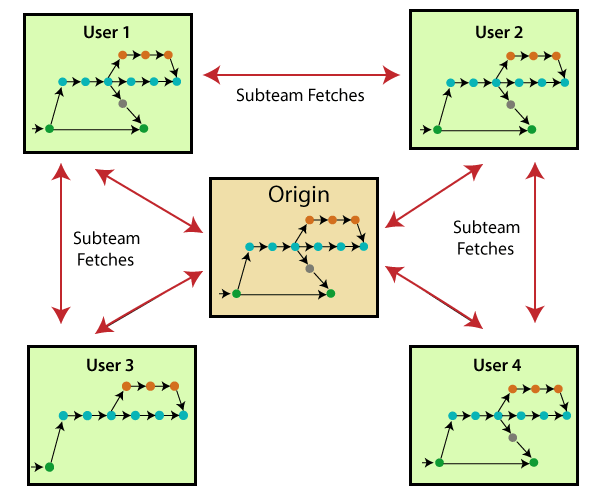
Some remarkable features of Git are as follows:



* **Open Source**

Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.

* **Scalable**  
  Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.
* **Distributed**  
  One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.



* **Security**  
  Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.
* **Speed**  
  Git is very **fast**, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a **huge speed**. Also, a centralized version control system continually communicates with a server somewhere.
* Performance tests conducted by Mozilla showed that it was **extremely fast compared to other VCSs**. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The **core part of Git**is **written in C**, which **ignores** runtime overheads associated with other high-level languages.

Git was developed to work on the Linux kernel; therefore, it is **capable** enough to **handle large** **repositories** effectively.

From the beginning, **speed** and **performance** have been Git's primary goals.

* **Supports non-linear development**

Git supports **seamless branching and merging**, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

* **Branching and Merging**

**Branching and merging** are the **great feature**s of Git, which makes it different from the other SCM tools. Git allows the **creation of multiple branches** without affecting each other. We can perform tasks like **creation**, **deletion**, and **merging** on branches, and these tasks take a few seconds only. Below are some features that can be achieved by branching:

* + We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
  + We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
  + We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.
  + The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.
* **Data Assurance**

The Git data model ensures the **cryptographic integrity** of every unit of our project. It provides a **unique commit ID** to every commit through a **SHA algorithm**. We can **retrieve** and **update** the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default.

* **Maintain the clean history**

Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

* **2.3 Version Control System**

**Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS −

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

## Distributed Version Control System

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

## Advantages of Git:-

### Free and open source

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

### Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

### Implicit backup

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

### Security

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

### No need of powerful hardware

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.

### Easier branching

CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

## DVCS Terminologies

## Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

## Working Directory and Staging Area or Index

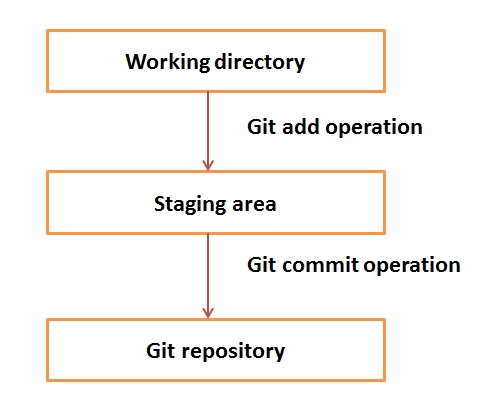
The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** − You modify a file from the working directory.

**Step 2** − You add these files to the staging area.

**Step 3** − You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

*# First commit*

[bash]$ git add sort.c

*# adds file to the staging area*

[bash]$ git commit –m “Added sort operation”

*# Second commit*

[bash]$ git add search.c

*# adds file to the staging area*

[bash]$ git commit –m “Added search operation”

* **Staging Area**

The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.



Another feature of Git that makes it apart from other SCM tools is that it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.

**Benefits of Git**

A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.

Some **significant benefits** of using Git are as follows:



* **Saves Time**

Git is lightning fast technology. Each command takes only a few seconds to execute so we can save a lot of time as compared to login to a GitHub account and find out its features.

* **Offline Working**

One of the most important benefits of Git is that it supports offline working. If we are facing internet connectivity issues, it will not affect our work. In Git, we can do almost everything locally. Comparatively, other CVS like SVN is limited and prefer the connection with the central repository.

* **Undo Mistakes**

One additional benefit of Git is we can Undo mistakes. Sometimes the undo can be a savior option for us. Git provides the undo option for almost everything.

* **Track the Changes**

Git facilitates with some exciting features such as Diff, Log, and Status, which allows us to track changes so we can check the status, compare our files or branches.

Why Git?

We have discussed many features and benefits of Git that demonstrate the undoubtedly Git as the leading version control system. Now, we will discuss some other points about why should we choose Git.



* **Git Integrity**

Git is developed to ensure the security and integrity of content being version controlled. It uses checksum during transit or tampering with the file system to confirm that information is not lost. Internally it creates a checksum value from the contents of the file and then verifies it when transmitting or storing data.

* **Trendy Version Control System**

Git is the most widely used version control system. It has maximum projects among all the version control systems. Due to its amazing workflow and features, it is a preferred choice of developers.

* **Everything is Local**

Almost All operations of Git can be performed locally; this is a significant reason for the use of Git. We will not have to ensure internet connectivity.

* **Collaborate to Public Projects**

There are many public projects available on the GitHub. We can collaborate on those projects and show our creativity to the world. Many developers are collaborating on public projects. The collaboration allows us to stand with experienced developers and learn a lot from them; thus, it takes our programming skills to the next level.

* **2.5 A Short History of GIT :-**

The Linux kernel is an open source software project of fairly large scope. During the early years of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool’s free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

* Speed
* Simple design
* Strong support for non-linear development (thousands of parallel branches)
* Fully distributed
* Able to handle large projects like the Linux kernel efficiently (speed and data size)

Until April 2005 Linus Torvalds was using BitKeeper for version control of the Linux Kernel development. He had a large number of volunteer developers working on the Linux Kernel and their contributions had to be managed.

BitKeeper was a nice tool for managing the enormous contribution by the developers. The Linux developers used the tool for free after an agreement between the two parties as BitKeeper was a proprietary source control management system which means you had to pay for the use of the tool. There came a conflict of interest after Andrew Tridgell created an open-source client for accessing the Bitkeeper version control system by reverse-engineering the BitKeeper protocols. This caused the copyright holder to withdrawal the free-to-use policy that they had earlier agreed upon. Many developers of the Linux kernel gave up access to the BitKeeper.

Linux knew he had to act fast to replace the version control system that he knew and loved so he took a working vacation to decide on what to do as the current free-to-use version control systems could not solve his problems at the time. The result of his vacation was the birth of a new version control system named **Git**.

* **2.9 Installing Git on Linux :-**

Git is an open-source distributed version control system that is available for everyone at zero cost. It is designed to handle minor to major projects with speed and efficiency. It is developed to co-ordinate the work among programmers. The version control allows you to track and work together with your team members at the same workspace.

Git is the most common source code management (SCM) and covers more users than earlier VCS systems like SVN**. Let's understand how to install Git on your Ubuntu server.**

**Step1: Start the General OS and Package update**

First of all, we should start the general OS and package updates. To do so, run the below command:

*$ apt-get update*

Now we have started the general OS and package updates. After this, we will run the general updates on the server so that we can get started with installing Git. To do so, run the following commands:

**Step2: Install Git**

To install Git, run the below command:

*$ apt-get install git-core*

The above command will install the Git on your system, but it may ask you to confirm the download and installation.

**Step3: Confirm Git the installation**

To confirm the installation, press '**y**' key on the editor. Now, Git is installed and ready to use.

When the central installation done, first check to ensure the executable file is set up and accessible. The best way to do this is the git version command. It will be run as:

*$ git --version*

Output:

git version 2.24.0

**Step4: Configure the Git for the First use**

Now you can start using Git on your system. You can explore many features of the version control system. To go with Git, you have to configure the initial user access process. It can be done with the git config command.

Suppose I want to register a user whose user name is "jspmntc" and email address is "jspmntc@xyz", then it will be done as follows:

To register a username, run the below command:

*$ git config --global user.name "jspmntc"*

To register an email address for the given author, run the below command:

*$ git config --global user.email "jspmntc@xyz"*

Now, you have successfully registered a user for the version control system.

It's important to understand that the **git config** tool works on a user according to the user. For example, if we have a user "**john**" registered on Git. Then there can be another user "**Mike**" on the same machine registered on Git. To do this, Mike must run the same command from his user account. The commits made by both the users will be done under their details in Git

# 2.10 Installing Git on Windows:-

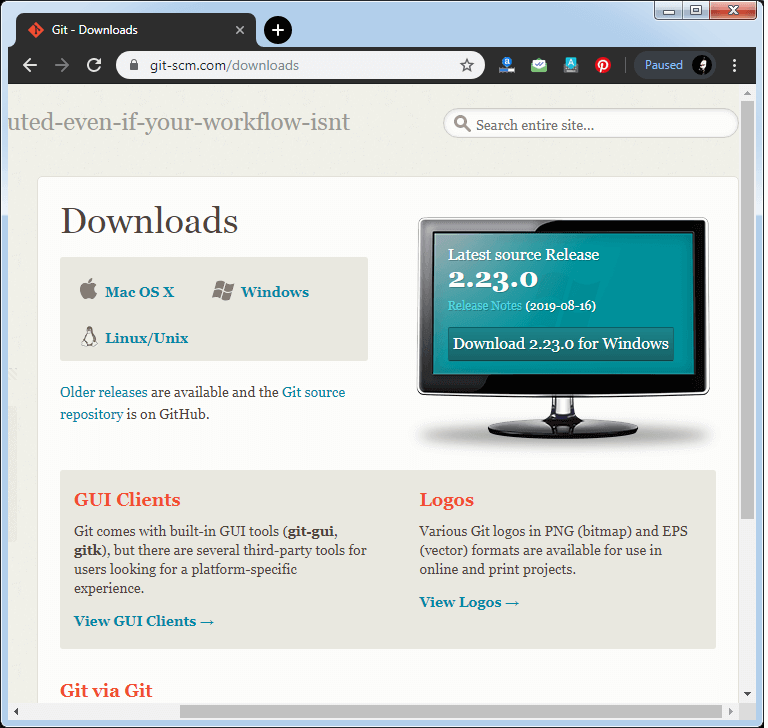
To use Git, you have to install it on your computer. Even if you have already installed Git, it's probably a good idea to upgrade it to the latest version. You can either install it as a package or via another installer or download it from its official site.

Now the question arises that how to download the Git installer package. Below is the stepwise installation process that helps you to download and install the Git.

## How to download Git?

**Step1**

To download the Git installer, visit the Git's official site and go to download page. The link for the download page is <https://git-scm.com/downloads>. The page looks like as



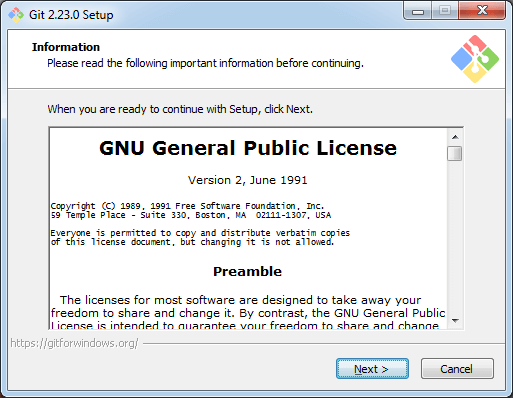
Click on the package given on the page as **download 2.36.1 for windows**. The download will start after selecting the package.

Now, the Git installer package has been downloaded.

## Install Git

**Step2**

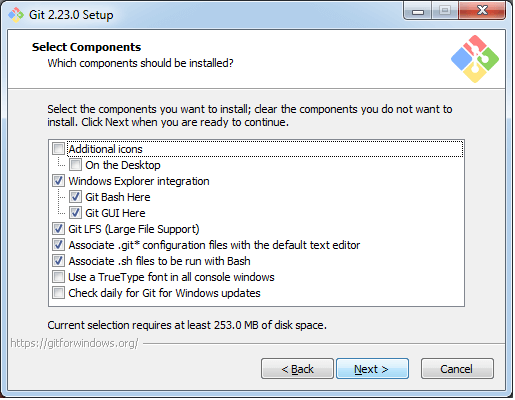
Click on the downloaded installer file and select **yes** to continue. After the selecting **yes** the installation begins, and the screen will look like as



Click on **next** to continue.

**Step3**

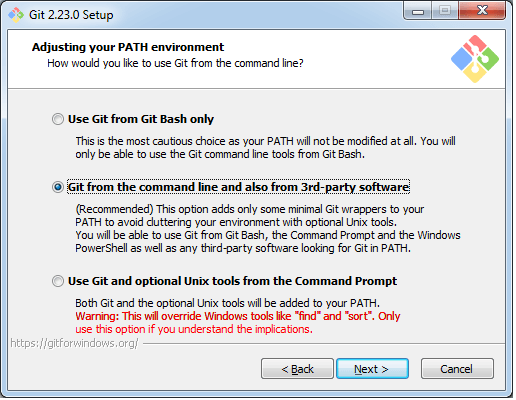
Default components are automatically selected in this step. You can also choose your required part.



Click next to continue.

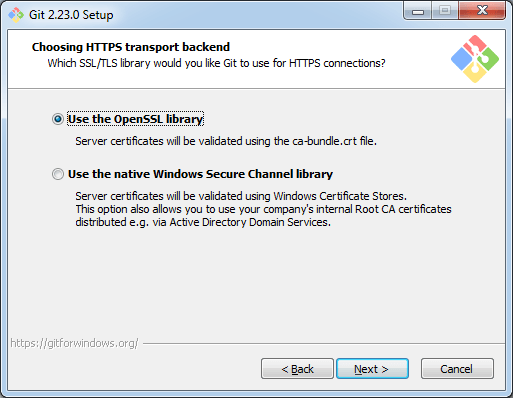
**Step4**

The default Git command-line options are selected automatically. You can choose your preferred choice. Click **next** to continue.



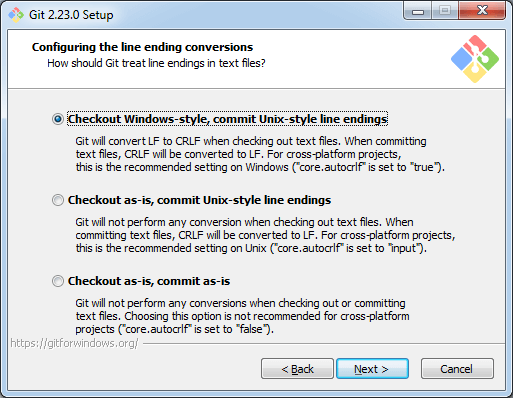
**Step5**

The default transport backend options are selected in this step. Click **next** to continue.



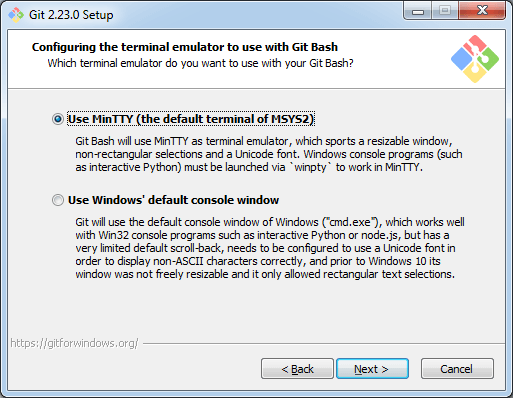
**Step6**

Select your required line ending option and click next to continue.



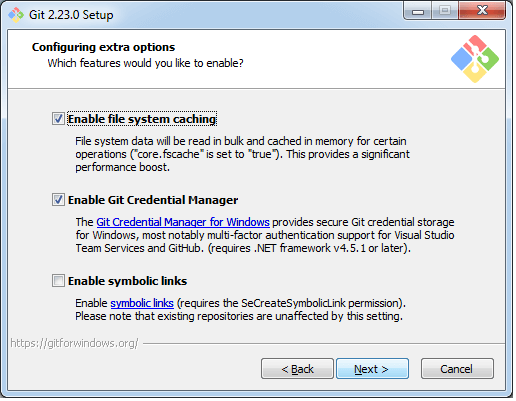
**Step7**

Select preferred terminal emulator clicks on the **next** to continue.



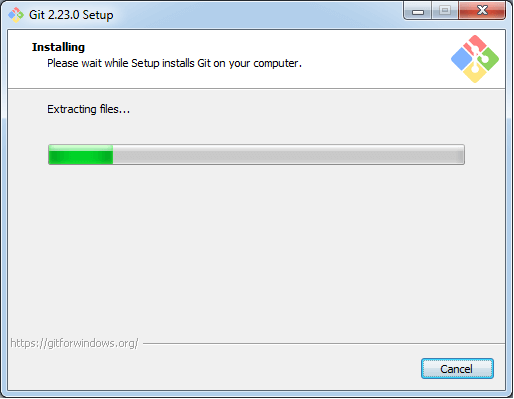
**Step8**

This is the last step that provides some extra features like system caching, credential management and symbolic link. Select the required features and click on the **next** option.



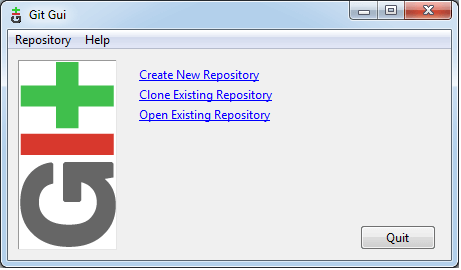
**Step9**

The files are being extracted in this step.



Therefore, The Git installation is completed. Now you can access the **Git Gui** and **Git Bash**.

The **Git Gui** looks like as



It facilitates with three features.

* Create New Repository
* Clone Existing Repository
* Open Existing Repository

The **Git Bash** looks like as



## Using Git with Command Line

To start using Git, we are first going to open up our Command shell.

For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:

$ git --version

git version 2.37.0.windows.1

## Configure Git

Now let Git know who you are. This is important for version control systems, as each Git commit uses this information:

### Example

$ git config --global user.name "TPC MCA"

$ git config --global user.email "tpc.mcajspmntc@gmail.com"

Change the user name and e-mail address to your own.

## Creating Git Folder

Now, let's create a new folder for our project:

$ mkdir myproject

$ cd myproject

mkdir **make**s a **new directory**.

cd **changes** the **current working directory**.

Now that we are in the correct directory. We can start by initializing Git!

## Initialize Git

Once you have navigated to the correct folder, you can initialize Git on that folder

$ git init

Initialized empty Git repository in C:/Users/MCA/myproject/.git/

You just created your first Git Repository!

## Git Adding New Files

You just created your first local Git repo. But it is empty.

So let's add some files, or create a new file using your favourite text editor. Then save or move it to the folder you just created.

For this example, to use a simple HTML file like this:

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>

</head>  
<body>  
  
<h1>Hello world!</h1>

<p>This is the first file in my new Git Repo.</p>

</body>  
</html>

and save it to our new folder myproject as index.html.

Let's go back to the terminal and list the files in our current working directory:

$ cd myproject

$ ls

index.html

Then we check the Git status and see if it is a part of our repo:

$ git status

*On branch master*

*No commits yet*

*Untracked files:*

*(use "git add ..." to include in what will be committed)     index.html nothing added to commit but untracked files present (use "git add" to track)*

Now Git is **aware** of the file, but has not **added** it to our repository!

Files in your Git repository folder can be in one of 2 states:

* Tracked - files that Git knows about and are added to the repository
* Untracked - files that are in your working directory, but not added to the repository

 When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

## Git Staging Environment

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

**Staged** files are files that are ready to be **committed** to the repository you are working on. You will learn more about commit shortly.

For now, we are done working with index.html. So we can add it to the Staging Environment:

$ git add index.html

The file should be **Staged**. Let's check the status:

$ git status

On branch master

No commits yet

Changes to be committed:

  (use "git rm --cached ..." to unstage)     new file: index.html

## Git Add More than One File

You can also stage more than one file at a time. Let's add 1 more files to our working folder. Use the text editor again.

A basic external style sheet (bluestyle.css):

### Example

Body

{  
background-color: lightblue;

}  
  
h1

{  
color: navy;

margin-left: 20px;

}

And update index.html to include the stylesheet:

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>

<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>

<p>This is the first file in my new Git Repo.</p>

</body>  
</html>

Now add all files in the current directory to the Staging Environment:

### Example

$ git add --all

Using --all instead of individual filenames will stage all changes (new, modified, and deleted) files.

### Example

git status

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached ..." to unstage)

new file: bluestyle.css

new file: index.html

**Note:** The shorthand command for git add --all is git add –A

## Git Commit

Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should **always** include a **message**.

By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

### Example

git commit -m "First release of Hello World!"

[master (root-commit) 221ec6e] First release of Hello World!

2 files changed, 26 insertions(+)

create mode 100644 bluestyle.css

create mode 100644 index.html

The commit command performs a commit, and the -m "*message*" adds a message.

The Staging Environment has been committed to our repo, with the message:  
"First release of Hello World!"

## Git Commit without Stage

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The -a option will automatically stage every changed, already tracked file.

Let's add a small update to index.html:

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>

<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
<h1>Hello world!</h1>

<p>This is the first file in my new Git Repo.</p>

<p>A new line in our file!</p>

</body>  
</html>

And check the status of our repository. But this time, we will use the --short option to see the changes in a more compact way:

### Example

git status --short

M index.html

**Note:** Short status flags are:

* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

We see the file we expected is modified. So let's commit it directly:

### Example

* git commit -a -m "Updated index.html with a new line"
* [master 09f4acd] Updated index.html with a new line
* 1 file changed, 1 insertion(+)

***{Warning:****Skipping the Staging Environment is not generally recommended. Skipping the stage step can sometimes make you include unwanted changes.****}***

## Git Commit Log

To view the history of commits for a repository, you can use the log command:

### Example

$ git log

* **2.13 Git Repository :-**

In Git, the repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repository in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

**Getting a Git Repository**

There are two ways to obtain a repository. They are as follows:

* Create a local repository and make it as Git repository.
* Clone a remote repository (already exists on a server).

In either case, you can start working on a Git repository.

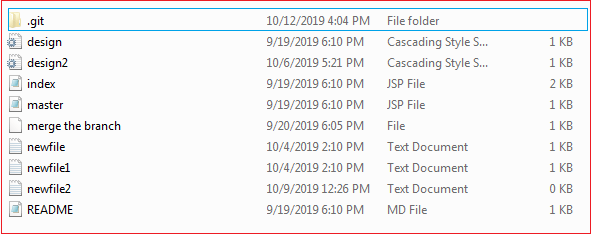
**Initializing a Repository**

If you want to share your project on a version control system and control it with Git. Then, browse your project's directory and start the git command line (Git Bash for Windows) here. To initialize a new repository, run the below command:

**Syntax:**

$ git init

The above command will create a new subdirectory named .git that holds all necessary repository files. The **.git** subdirectory can be understood as a Git repository skeleton. Consider the below image:



An empty repository .git is added to my existing project. If we want to start version-controlling for existing files, we should track these files with git add command, followed by a commit.

We can list all the untracked files by git status command.

$ git status

he list of all untracked files is displayed by the git status command. To share these files on the version control system, we have to track it with git add command followed by a commit. To track the files, operate git add command as follows:

**Syntax:**

$ git add **<filename>**

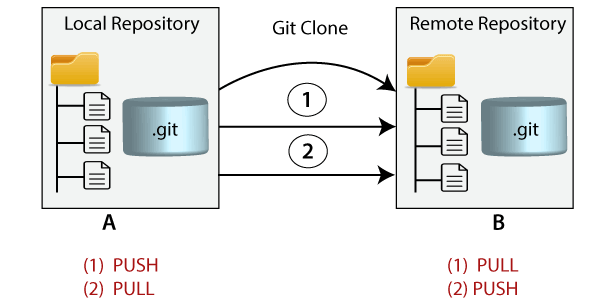
To commit a file, perform the git commit command as follows:

$ git commit -m "Commit message."

We can also create new files. To share the new file, follow the same procedure as described above; add and commit it for sharing. Now, you have a repository to share.

* **2.14 Git Clone**

In Git, cloning is the act of making a copy of any target repository. The target repository can be remote or local. You can clone your repository from the remote repository to create a local copy on your system. Also, you can sync between the two locations.



**Git Clone Command**

The **git clone** is a command-line utility which is used to make a local copy of a remote repository. It accesses the repository through a remote URL.

Usually, the original repository is located on a remote server, often from a Git service like GitHub, Bitbucket, or GitLab. The remote repository URL is referred to the **origin**.

**Syntax:**

$ git clone **<repository** URL**>**

**Git Clone Repository**

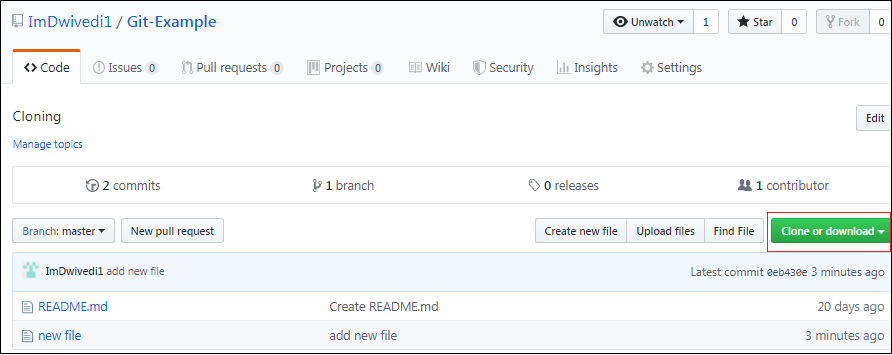
Suppose, you want to clone a repository from GitHub, or have an existing repository owned by any other user you would like to contribute. Steps to clone a repository are as follows:

**Step 1:**

Open GitHub and navigate to the main page of the repository.

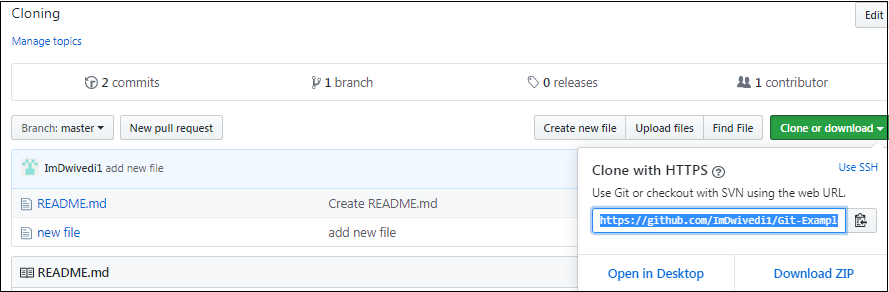
**Step 2:**

Under the repository name, click on **Clone or download**.



**Step 3:**

Select the **Clone with HTTPs section** and **copy the clone URL** for the repository. For the empty repository, you can copy the repository page URL from your browser and skip to next step.



**Step 4:**

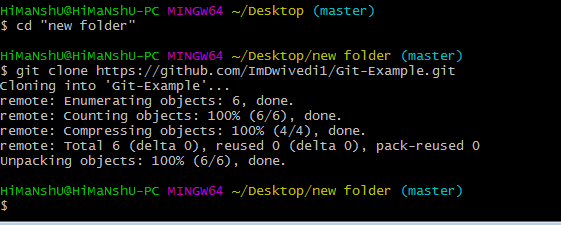
Open Git Bash and change the current working directory to your desired location where you want to create the local copy of the repository.

**Step 5:**

Use the git clone command with repository URL to make a copy of the remote repository. See the below command:

$ git clone https://github.com/ImDwivedi1/Git-Example.git

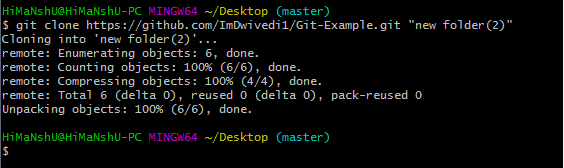
Now, Press Enter. Hence, your local cloned repository will be created. See the below output:



**Cloning a Repository into a Specific Local Folder**

Git allows cloning the repository into a specific directory without switching to that particular directory. You can specify that directory as the next command-line argument in git clone command. See the below command:

$ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"



The given command does the same thing as the previous one, but the target directory is switched to the specified directory.

**Git Clone Branch**

Git allows making a copy of only a particular branch from a repository. You can make a directory for the individual branch by using the git clone command. To make a clone branch, you need to specify the branch name with -b command. Below is the syntax of the command to clone the specific git branch:

**Syntax:**

$ git clone -b **<Branch** name**><Repository** URL**>**

See the below command:

$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"



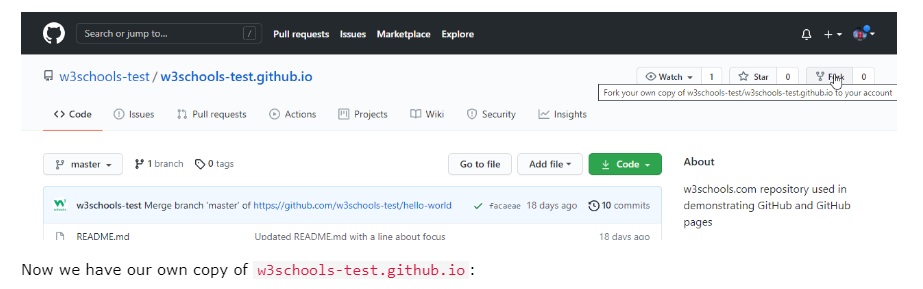
In the given output, only the master branch is cloned from the principal repository Git-Example.

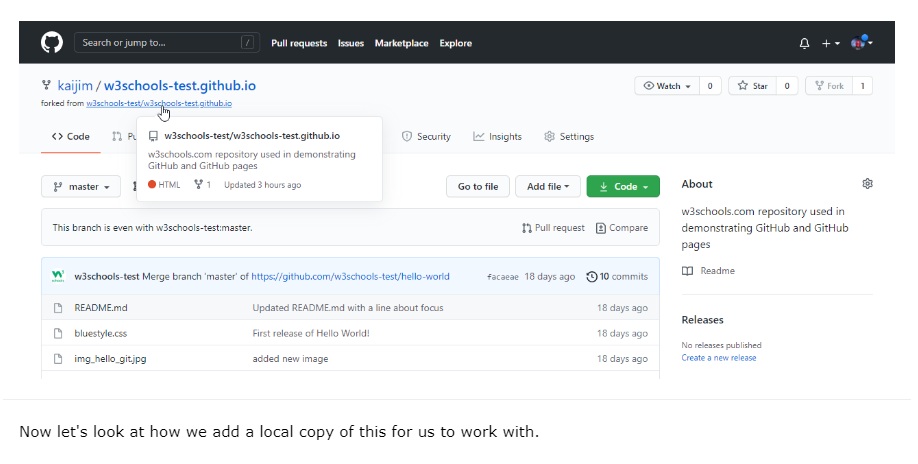
## Fork a Repository

## At the heart of Git is collaboration. However, Git does not allow you to add code to someone else's repository without access rights.

A fork is a copy of a repository. This is useful when you want to contribute to someone else's project or start your own project based on theirs.

fork is not a command in Git, but something offered in GitHub and other repository hosts. Let's start by logging in to GitHub, and fork our repository:  
https://github.com/w3schools-test/w3schools-test.github.io

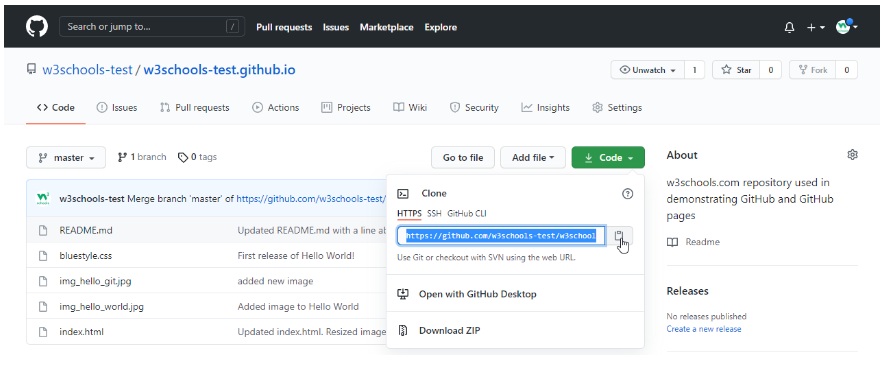




Now we have our own fork, but only on GitHub. We also want a clone on our local Git to keep working on it.

A clone is a full copy of a repository, including all logging and versions of files.

Move back to the **original** repository, and click the green "Code" button to get the URL to clone:



Open your Git bash and clone the repository:

### Example

git clone https://github.com/w3schools-test/w3schools-test.github.io.git

Cloning into 'w3schools-test.github.io'...

remote: Enumerating objects: 33, done.

remote: Counting objects: 100% (33/33), done.

remote: Compressing objects: 100% (15/15), done.

remote: Total 33 (delta 18), reused 33 (delta 18), pack-reused 0

Receiving objects: 100% (33/33), 94.79 KiB | 3.16 MiB/s, done.

Resolving deltas: 100% (18/18), done.

Take a look in your file system, and you will see a new directory named after the cloned project:

### Example

ls

w3schools-test.github.io/

**Note:** To specify a specific folder to clone to, add the name of the folder after the repository URL, like this: git clone

https://github.com/w3schools-test/w3schools-test.github.io.git myfolder

Navigate to the new directory, and check the status:

### Example

cd w3schools-test.github.io

git status

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

And check the log to confirm that we have the full repository data:

### Example

git log

commit facaeae8fd87dcb63629f108f401aa9c3614d4e6 (HEAD -> master, origin/master, origin/HEAD)

Merge: e7de78f 5a04b6f

Author: w3schools-test

Date: Fri Mar 26 15:44:10 2021 +0100

Merge branch 'master' of https://github.com/w3schools-test/hello-world

commit e7de78fdefdda51f6f961829fcbdf197e9b926b6

Author: w3schools-test

Date: Fri Mar 26 15:37:22 2021 +0100

Updated index.html. Resized image

.....

Now we have a full copy of the original repository.

## Configuring Remotes

Basically, we have a full copy of a repository, whose origin we are not allowed to make changes to.

Let's see how the remotes of this Git is set up:

### Example

git remote -v

origin https://github.com/w3schools-test/w3schools-test.github.io.git (fetch)

origin https://github.com/w3schools-test/w3schools-test.github.io.git (push)

We see that origin is set up to the original "w3schools-test" repository, we also want to add our own fork.

First, we rename the original origin remote:

### Example

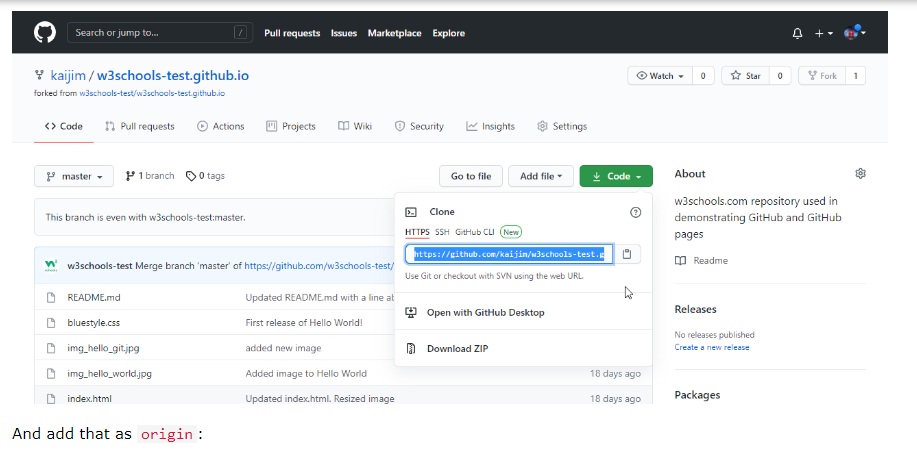
git remote rename origin upstream

git remote -v

upstream https://github.com/w3schools-test/w3schools-test.github.io.git (fetch)

upstream https://github.com/w3schools-test/w3schools-test.github.io.git (push)

Then fetch the URL of our own fork:



### Example

git remote add origin https://github.com/kaijim/w3schools-test.github.io.git

git remote -v

origin https://github.com/kaijim/w3schools-test.github.io.git (fetch)

origin https://github.com/kaijim/w3schools-test.github.io.git (push)

upstream https://github.com/w3schools-test/w3schools-test.github.io.git (fetch)

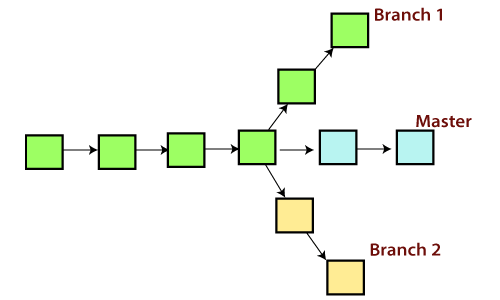
upstream https://github.com/w3schools-test/w3schools-test.github.io.git (push)

**Note:** According to Git naming conventions, it is recommended to name your own repository origin, and the one you forked for upstream

Now we have 2 remotes:

* origin - our own fork, where we have read and write access
* upstream - the original, where we have read-only access
* **2.16 Git Branching :-**

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



## Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

## Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

**The Operations that can be performed on a branch:**

### Create Branch

You can create a new branch with the help of the **git branch** command. This command will be used as:

**Syntax:**

$ git branch  **<branch** name**>**

For example :- $ git branch  **branch B1**

This command will create the **branch B1** locally in Git directory.

### List Branch

You can List all of the available branches in your repository by using the following command.

Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

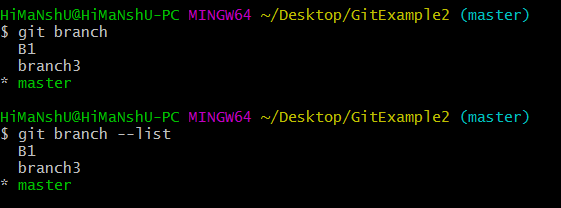
**Syntax:**

$ git branch --list

**or**

$ git branch

**Output:**



  Here, both commands are listing the available branches in the repository. The symbol **\*** is representing currently active branch.

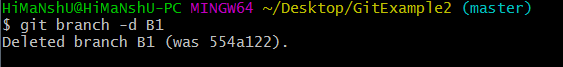
### Delete Branch

You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

**Syntax:**

$ git branch -d**<branch** name**>**

**Output:**



This command will delete the existing branch B1 from the repository.

The **git branch d** command can be used in two formats. Another format of this command is **git branch D**. The '**git branch D**' command is used to delete the specified branch.

$ git branch -D **<branch** name**>**

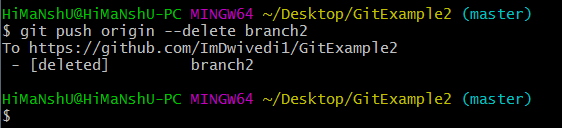
### Delete a Remote Branch

You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

**Syntax:**

$ git push origin -delete **<branch** name**>**

**Output:**



As you can see in the above output, the remote branch named **branch2** from my GitHub account is deleted.

### Switch Branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

$ git checkout**<branch** name**>**

**Switch from master Branch**

You can switch from master to any other branch available on your repository without making any commit.

**Syntax:**

$ git checkout **<branch** name**>**

**Output:**

Git Branch

As you can see in the output, branches are switched from **master** to **branch4** without making any commit.

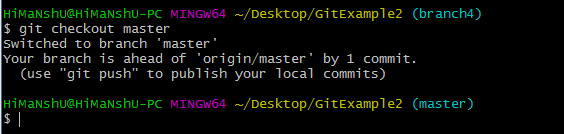
**Switch to master branch**

You can switch to the master branch from any other branch with the help of below command.

**Syntax:**

$ git branch -m master

**Output:**



As you can see in the above output, branches are switched from **branch1 to master** without making any commit.

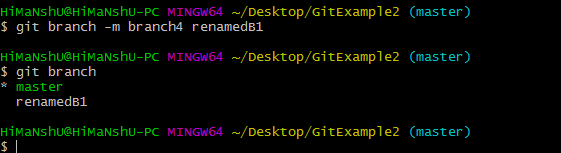
### Rename Branch

We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

**Syntax:**

$ git branch -m **<old** branch name**><new** branch name**>**

**Output:**



As you can see in the above output, **branch4** renamed as **renamedB1**.

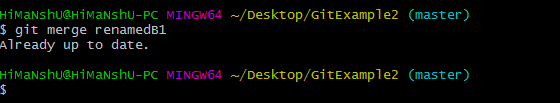
### Merge Branch

Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

**Syntax:**

$ git merge **<branch** name**>**

**Output:**



From the above output, you can see that **the master** branch **merged** with **renamedB1**. Since I have made no-commit before merging, so the output is showing as already up to date.

**In Git, a branch is a new/separate version of the main repository.**

Let's say you have a large project, and you need to update the design on it.

How would that work without and with Git:

**Without Git:**

* Make copies of all the relevant files to avoid impacting the live version
* Start working with the design and find that code depend on code in other files, that also need to be changed!
* Make copies of the dependant files as well. Making sure that every file dependency references the correct file name
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Save all your files, making a note of the names of the copies you were working on
* Work on the unrelated error and update the code to fix it
* Go back to the design, and finish the work there
* Copy the code or rename the files, so the updated design is on the live version
* (2 weeks later, you realize that the unrelated error was not fixed in the new design version because you copied the files before the fix)

**With Git:**

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix
* Fix the unrelated error and merge the small-error-fix branch with the main branch
* You go back to the new-design branch, and finish the work there
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

**New Git Branch**

Let add some new features to our index.html page.

We are working in our local repository, and we do not want to disturb or possibly wreck the main project.

So we create a new branch:

### Example

git branch hello-world-images

Now we created a new branch called "hello-world-images"

Let's confirm that we have created a new branch:

### Example

git branch

hello-world-images

\* master

We can see the new branch with the name "hello-world-images", but the \* beside master specifies that we are currently on that branch.

checkout is the command used to check out a branch. Moving us **from** the current branch, **to** the one specified at the end of the command:

### Example

git checkout hello-world-images

Switched to branch 'hello-world-images'

Now we have moved our current workspace from the master branch, to the new branch

Open your favourite editor and make some changes.

For this example, we added an image (img\_hello\_world.jpg) to the working folder and a line of code in the index.html file:

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<div><img src="img\_hello\_world.jpg" alt="Hello World from Space"  
style="width:100%;max-width:960px"></div>  
<p>This is the first file in my new Git Repo.</p>  
<p>A new line in our file!</p>  
  
</body>  
</html>

We have made changes to a file and added a new file in the working directory (same directory as the main branch).

Now check the status of the current branch:

### Example

git status

On branch hello-world-images

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git restore ..." to discard changes in working directory)

modified: index.html

Untracked files:

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

img\_hello\_world.jpg

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

So let's go through what happens here:

* There are changes to our index.html, but the file is not staged for commit
* img\_hello\_world.jpg is not tracked

So we need to add both files to the Staging Environment for this branch:

### Example

git add --all

Using --all instead of individual filenames will **Stage** all changed (new, modified, and deleted) files.

Check the status of the branch:

### Example

git status

On branch hello-world-images

Changes to be committed:

  (use "git restore --staged ..." to unstage)

    new file: img\_hello\_world.jpg

    modified: index.html

We are happy with our changes. So we will commit them to the branch:

### Example

git commit -m "Added image to Hello World"

[hello-world-images 0312c55] Added image to Hello World

2 files changed, 1 insertion(+)

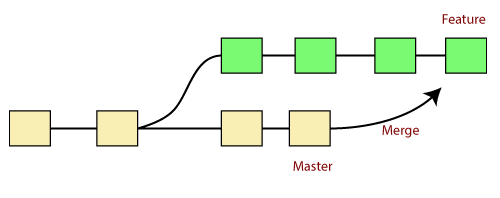
create mode 100644 img\_hello\_world.jpg

Now we have a new branch, that is different from the master branch.

**Note:** Using the -b option on checkout will create a new branch, and move to it, if it does not exist

* **2.17 Git Merge and Merge Conflict**

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



It is used to maintain distinct lines of development; at some stage, you want to merge the changes in one branch. It is essential to understand how merging works in Git.

In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

## The "git merge" command

The git merge command is used to merge the branches.

The syntax for the git merge command is as:

$ git merge **<query>**

It can be used in various context. Some are as follows:

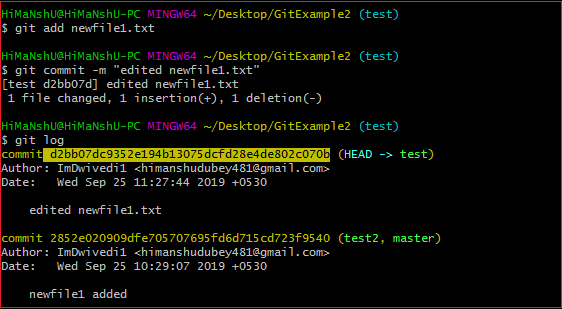
**Scenario1: To merge the specified commit to currently active branch:**

Use the below command to merge the specified commit to currently active branch.

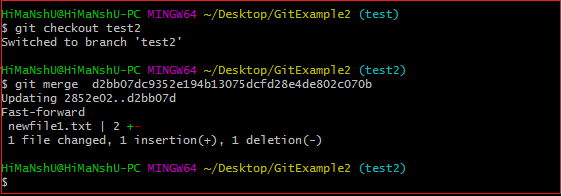
$ git merge **<commit>**

The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.



Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:



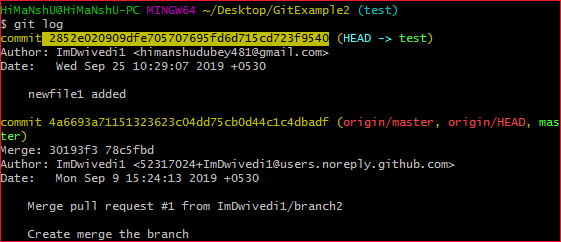
In the above output, we have merged the previous commit in the active branch test2.

**Scenario2: To merge commits into the master branch:**

To merge a specified commit into master, first discover its commit id. Use the log command to find the particular commit id.

$git log

See the below output:



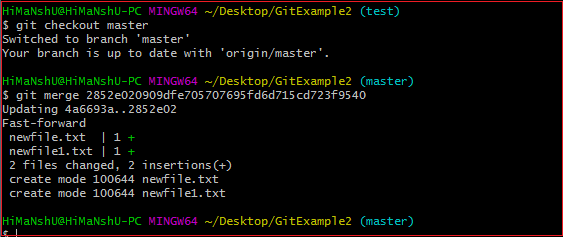
To merge the commits into the master branch, switch over to the master branch.

$ git checkout master

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name. The syntax for this is as follows:

$ git merge master

See the below output:



As shown in the above output, the commit for the commit id **2852e020909dfe705707695fd6d715cd723f9540** has merged into the master branch. Two files have changed in master branch. However, we have made this commit in the **test** branch. So, it is possible to merge any commit in any of the branches.

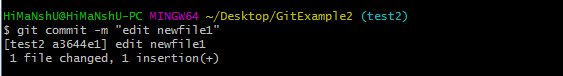
Open new files, and you will notice that the new line that we have committed to the test branch is now copied on the master branch.

**Scenario 3: Git merge branch.**

Git allows merging the whole branch in another branch. Suppose you have made many changes on a branch and want to merge all of that at a time. Git allows you to do so. See the below example:

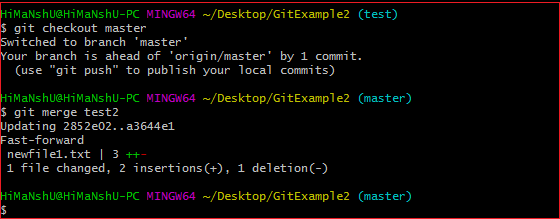
Git Merge and Merge Conflict

In the given output, I have made changes in newfile1 on the test branch. Now, I have committed this change in the test branch.



Now, switch to the desired branch you want to merge. In the given example, I have switched to the master branch. Perform the below command to merge the whole branch in the active branch.

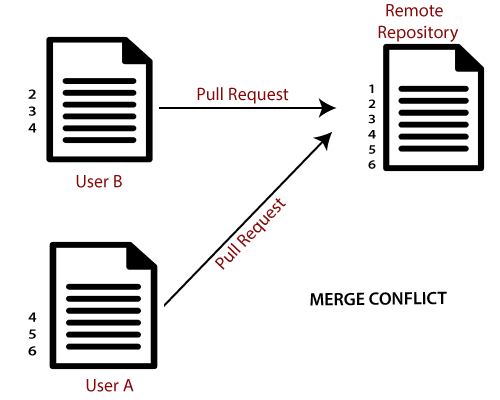
$ git merge **<branchname>**



As you can see from the given output, the whole commits of branch test2 have merged to branch master.

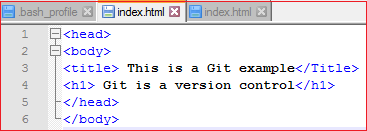
## Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



Let's understand it by an example.

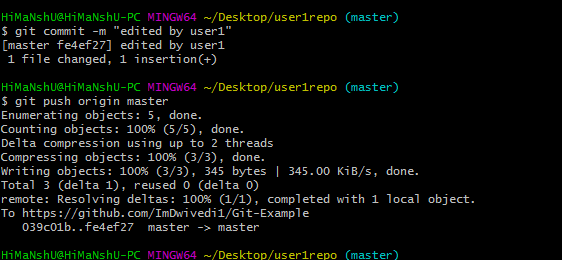
Suppose my remote repository has cloned by two of my team member **user1** and **user2**. The user1 made changes as below in my projects index file.



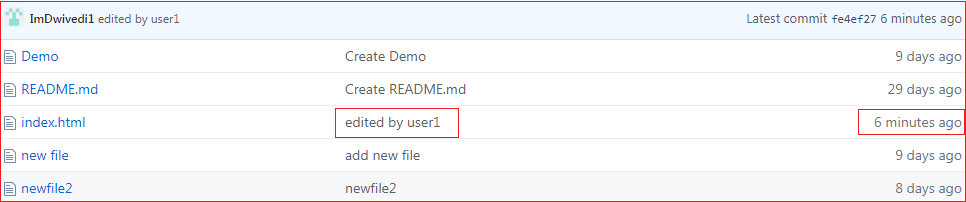
Update it in the local repository with the help of git add command.

Git Merge and Merge Conflict

Now commit the changes and update it with the remote repository. See the below output:

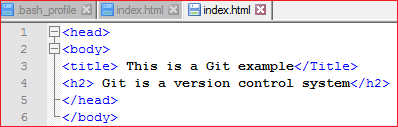


Now, my remote repository will look like this:

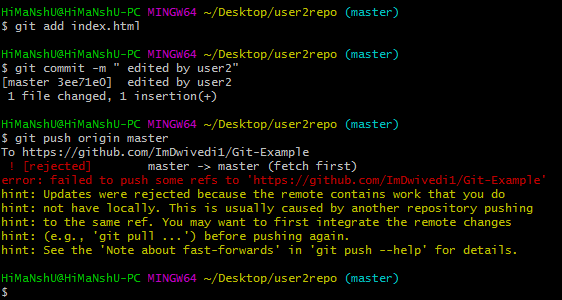


It will show the status of the file like edited by whom and when.

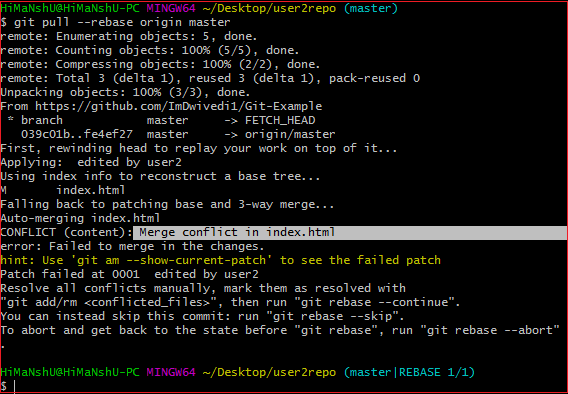
Now, at the same time, **user2** also update the index file as follows.



User2 has added and committed the changes in the local repository. But when he tries to push it to remote server, it will throw errors. See the below output:



In the above output, the server knows that the file is already updated and not merged with other branches. So, the push request was rejected by the remote server. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**. It will suggest you to pull the repository first before the push. See the below command:



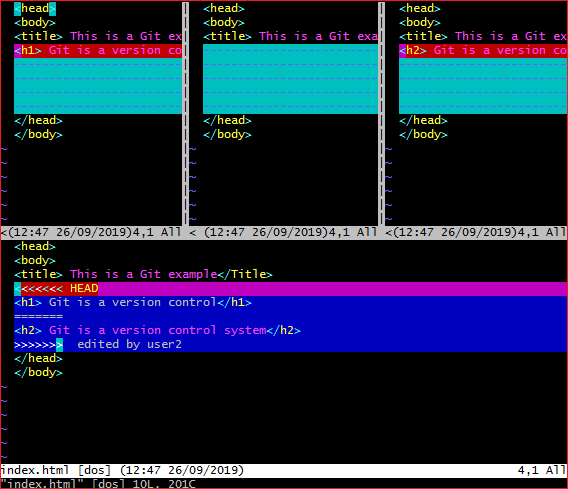
In the given output, git rebase command is used to pull the repository from the remote URL. Here, it will show the error message like **merge conflict in <filename>**.

## Resolve Conflict:

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

$ git mergetool

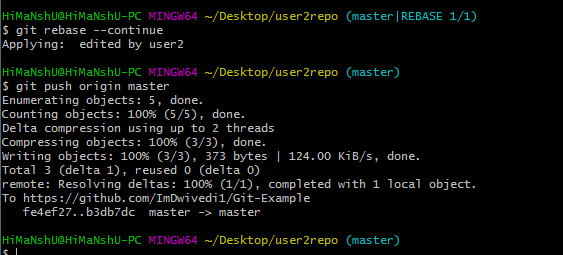
In my repository, it will result in:



The above output shows the status of the conflicted file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the rebase command. It will be used as follows:

$ git rebase --continue

Hence, the conflict has resolved. See the below output:



In the above output, the conflict has resolved, and the local repository is synchronized with a remote repository.

To see that which is the first edited text of the merge conflict in your file, search the file attached with conflict marker **<<<<<<<**. You can see the changes from the **HEAD** or base branch after the line **<<<<<<< HEAD** in your text editor. Next, you can see the divider like **=======**. It divides your changes from the changes in the other branch, **followed by >>>>>>> BRANCH-NAME**. In the above example, user1 wrote "<h1> Git is a version control</h1>" in the base or HEAD branch and user2 wrote "<h2> Git is a version control</h2>".

Decide whether you want to keep only your branch's changes or the other branch's changes, or create a new change. Delete the conflict markers **<<<<<<<, =======, >>>>>>>** and create final changes you want to merge.

## Merge Branches

We have the emergency fix ready, and so let's merge the master and emergency-fix branches.

First, we need to change to the master branch:

### Example

git checkout master

Switched to branch 'master'

Now we merge the current branch (master) with emergency-fix:

### Example

git merge emergency-fix

Updating 09f4acd..dfa79db

Fast-forward

index.html | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

Since the emergency-fix branch came directly from master, and no other changes had been made to master while we were working, Git sees this as a continuation of master. So it can "Fast-forward", just pointing both master and emergency-fix to the same commit.

As master and emergency-fix are essentially the same now, we can delete emergency-fix, as it is no longer needed:

### Example

git branch -d emergency-fix

Deleted branch emergency-fix (was dfa79db).

## Merge Conflict

Now we can move over to hello-world-images and keep working. Add another image file (img\_hello\_git.jpg) and change index.html, so it shows it:

### Example

git checkout hello-world-images

Switched to branch 'hello-world-images'

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<div><img src="img\_hello\_world.jpg" alt="Hello World from Space" style="width:100%;max-width:960px"></div>  
<p>This is the first file in my new Git Repo.</p>  
<p>A new line in our file!</p>  
<div><img src="img\_hello\_git.jpg" alt="Hello Git" style="width:100%;max-width:640px"></div>  
  
</body>  
</html>

Now, we are done with our work here and can stage and commit for this branch:

### Example

git add --all

git commit -m "added new image"

[hello-world-images 1f1584e] added new image

2 files changed, 1 insertion(+)

create mode 100644 img\_hello\_git.jpg

We see that index.html has been changed in both branches. Now we are ready to merge hello-world-images into master. But what will happen to the changes we recently made in master?

### Example

git checkout master

git merge hello-world-images

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

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

The merge failed, as there is conflict between the versions for index.html. Let us check the status:

### Example

git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Changes to be committed:

new file: img\_hello\_git.jpg

new file: img\_hello\_world.jpg

Unmerged paths:

(use "git add ..." to mark resolution)

both modified: index.html

This confirms there is a conflict in index.html, but the image files are ready and stagedto be committed.

So we need to fix that conflict. Open the file in our editor:

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<div><img src="img\_hello\_world.jpg" alt="Hello World from Space" style="width:100%;max-width:960px"></div>  
<p>This is the first file in my new Git Repo.</p>  
<<<<<<< HEAD  
<p>This line is here to show how merging works.</p>  
=======  
<p>A new line in our file!</p>  
<div><img src="img\_hello\_git.jpg" alt="Hello Git" style="width:100%;max-width:640px"></div>  
>>>>>>> hello-world-images  
  
</body>  
</html>

We can see the differences between the versions and edit it like we want:

### Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<div><img src="img\_hello\_world.jpg" alt="Hello World from Space" style="width:100%;max-width:960px"></div>  
<p>This is the first file in my new Git Repo.</p>  
<p>This line is here to show how merging works.</p>  
<div><img src="img\_hello\_git.jpg" alt="Hello Git" style="width:100%;max-width:640px"></div>  
  
</body>  
</html>

Now we can stage index.html and check the status:

### Example

git add index.html

git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

new file: img\_hello\_git.jpg

new file: img\_hello\_world.jpg

modified: index.html

The conflict has been fixed, and we can use commit to conclude the merge:

### Example

git commit -m "merged with hello-world-images after fixing conflicts"

[master e0b6038] merged with hello-world-images after fixing conflicts

And delete the hello-world-images branch:

### Example

git branch -d hello-world-images

Deleted branch hello-world-images (was 1f1584e).

Now you have a better understanding of how branches and merging works. Time to start working with a remote repository!