**Version Control System**

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

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

**Types of Version Control Systems**

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

**Centralized version control system (CVCS)**

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.

**Distributed Version Control System**

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the sever 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.

**Advantages of Git**

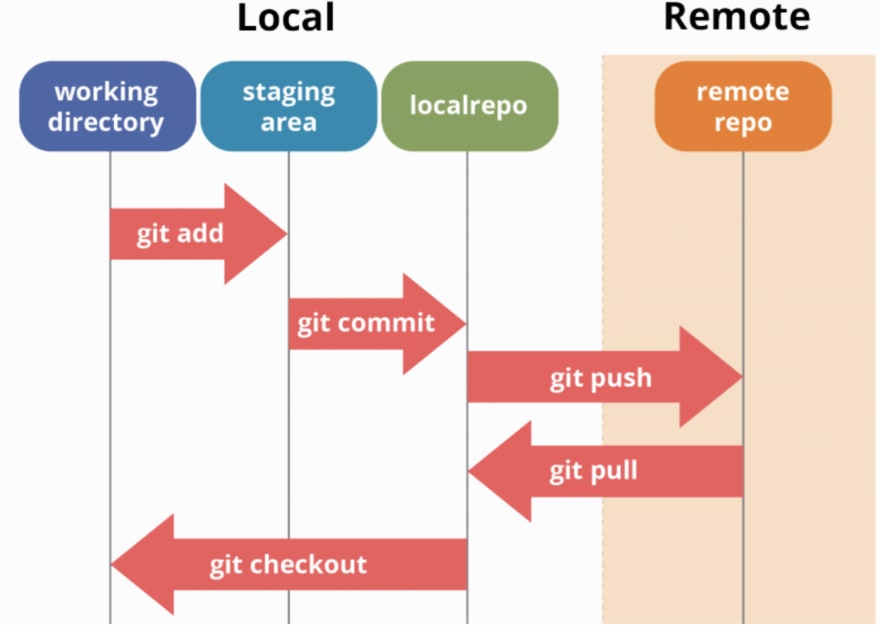
* Free and Open source.
* Fast and small
* Implicit backup
* Security
* No need of powerful hardware
* Easier branching

**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.

**Work Flow of Git**

1. You modify a file from the working directory.
2. You add these files to the staging area.
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.



**Git Installation**

Please refer below link for installing git

<https://phoenixnap.com/kb/how-to-install-git-windows#:~:text=Steps%20For%20Installing%20Git%20for%20Windows%201%20Extract,which%20extra%20option%20you%20would%20like%20to%20enable>.

**Setting up a repository**

**git init / git clone / git config**

**git init**

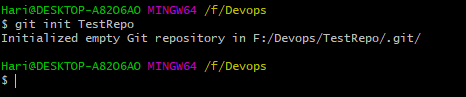
The git init command initializes a new, empty repository.



Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

Compared to SVN, the git init command is an incredibly easy way to create new version-controlled projects. Git doesn’t require you to create a repository, import files, and check out a working copy. Additionally, Git does not require any pre-existing server or admin privileges. All you have to do is cd into your project subdirectory and run git init, and you'll have a fully functional Git repository.

git init <directory>



**Git Files Directory**

How the directory structure is organized and where the components of the Git tree fall in to it.

* The .**git** directory holds the meat of your local repo. It holds subdirectories for objects, refs, config, index, HEAD, and more.
* The **objects** folder is where the three objects: commit, tree and blob are housed. Each object gets its own sub folder.
  + **Blob**

Blobs, on the other hand, are just contents — binary streams of data, every blob in git are identified by its SHA-1 hash. SHA-1 hashes consist of 20 bytes

* + **Tree**

A tree is a simple object that has a bunch of pointers to blobs and other trees - it generally represents the contents of a directory or subdirectory.

* + **Commit**

Commit set a message about the changes you were done. The commit also saves a revision of the code and you can revert the code to any version anytime.

* The **refs** folder holds references to all of the objects in the directory. Let's say you have some orphaned objects on branches that no longer exist. You can go into this folder and prune them off of your branch.
* The **config** folder contains all of your configurations including the credentials you use to connect to GitHub; your email and name that you used to configure your global config.
* The **HEAD** file is what contains the pointer or reference to your working directory or branch and its latest commit.
* The **index** file is for when you use git add to stage your files for a commit. They leave the index file after the commit.

**git add**

The **git add** command adds a file to the Git staging area. This area contains a list of all the files you have recently changed.



**git commit**

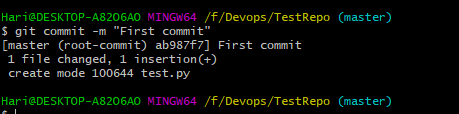
The **git commit** the staged snapshot.

**git commit –a**

Commita snapshot of all changes in the working directory.

**git commit –m**

A shortcut command that immediately creates a commit with a passed commit message.



**git commit -am "commit message"**

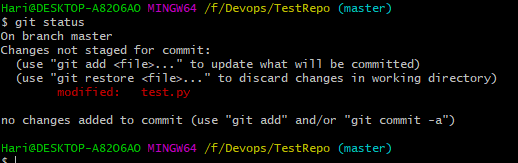
A power user shortcut command that combines the -a and -m options. This combination immediately creates a commit of all the staged changes and takes an inline commit message.

**git commit –amend**

This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit.

**git status**

The git status command displays the state of the working directory and the staging area.



**git log**

The git log command displays committed snapshots.

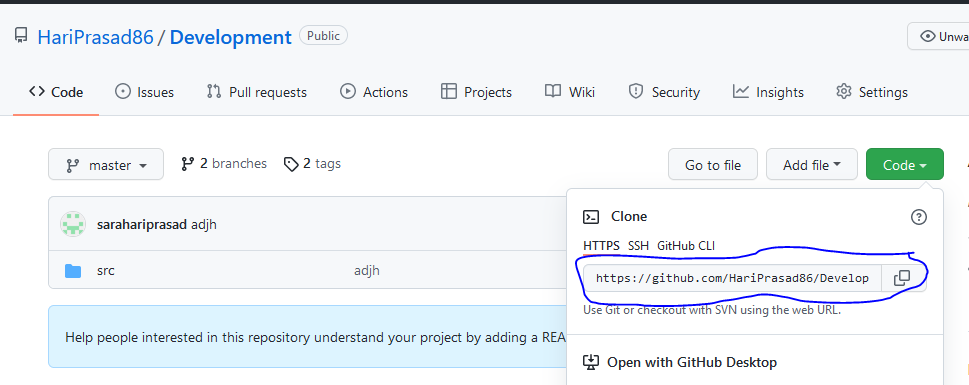


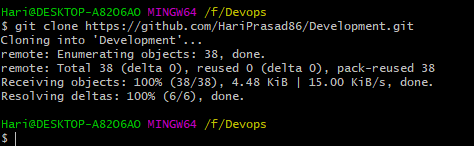
**Git clone**

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

git clone <repository URL>

Copy the git remote repository url like below screen

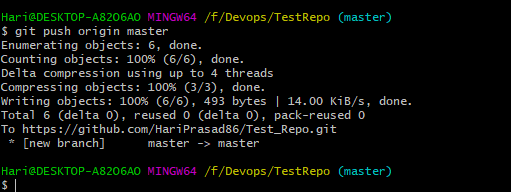




**Git push**

The git push command is used to upload local repository content to a remote repository.





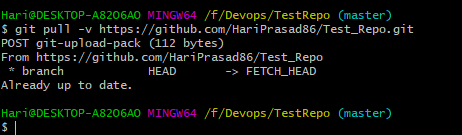
**Git pull**

The pull command is used to access the changes (commits)from a remote repository to the local repository.

$ **git pull <option> [<repository URL><refspec>...]**

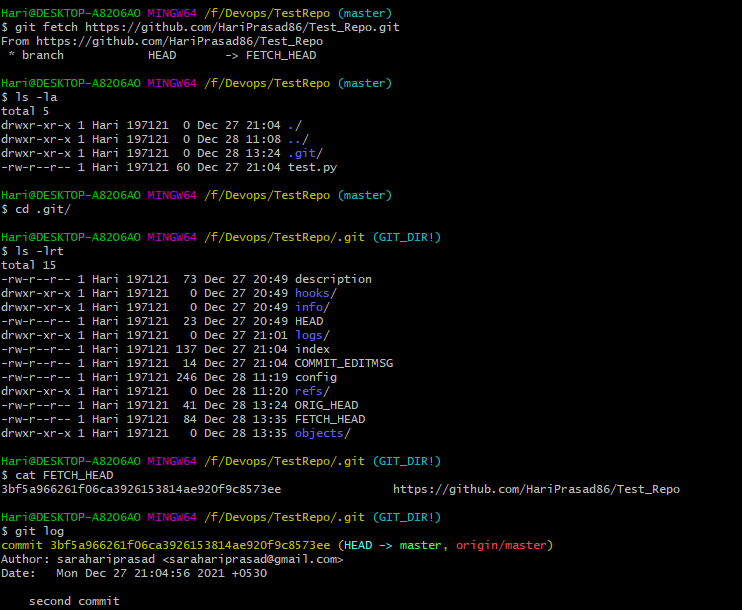
**<option>:** Options are the commands; these commands are used as an additional option in a particular command. Options can be -q (quiet), -v (verbose), -e(edit) and more.

**<repository URL>:** Repository URL is your remote repository's URL where you have stored your original repositories like GitHub or any other git service



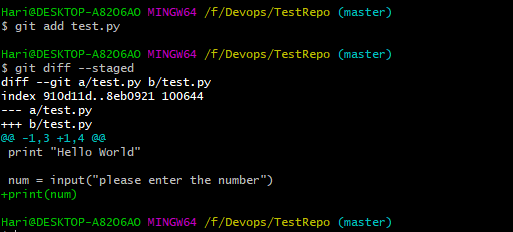
**Git fetch**

The git fetch command is used to download commits, files and references from a remote repository into the local repository. It is used to see what other members of the team have been working on.



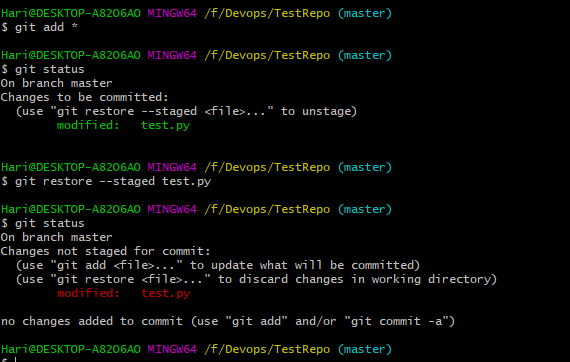
**Git diff**

The above command will display the changes of already staged files. Consider the below output:

****

**git restore**

The "**restore**" command helps to unstage or even discard uncommitted local changes.

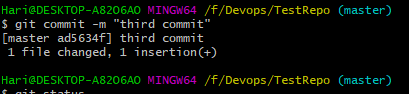


**Git revert**

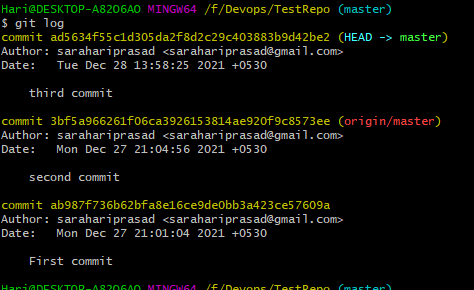
The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history.

Please find the below output screens.

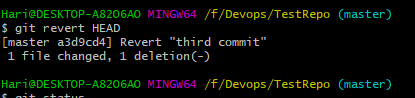
Step1. Commited new changes to local repository.



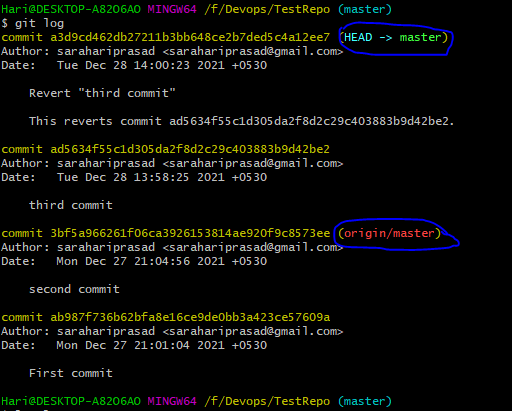
Step2. Check git log and find the latest commit.



Step 3. Reverting commit changes.



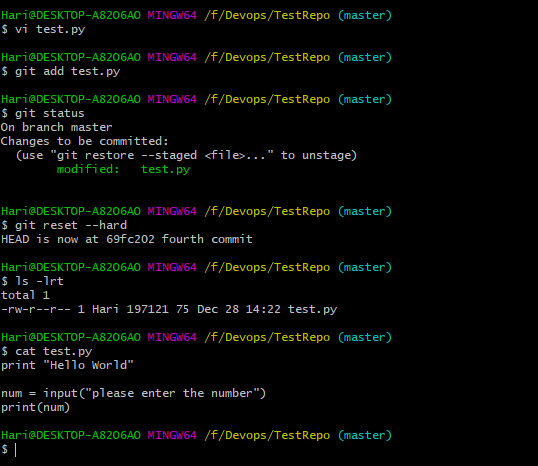
Step4. Check the git log again.



Finally we are at version c8573ee

**Git reset**

The **git reset** command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments --soft, --mixed, --hard. The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

****

**BRANCHING STRATEGIES**

Git branching strategies allow a code base to evolve organically in a coherent way. A branching strategy is a convention, or a set of rules, that describes when branches are created, naming guidelines for branches, what use branches should have, and so on

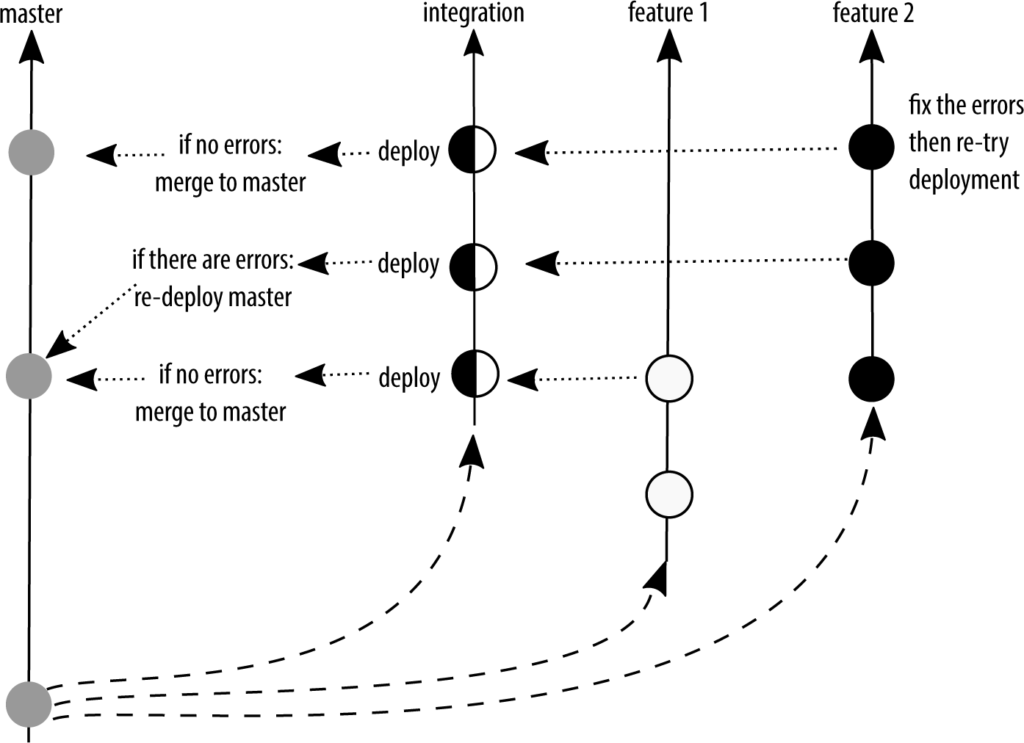
**Master branch** – The branch from which software is released to production. Only production ready code can be committed to this branch. All commits are tagged, since they represent releases.

**Develop/Dev branch** – The main branch that developers work on.

**Feature branches** – Feature branches are used to develop new features. Feature branches will be eventually merged back into the development branch. Each feature branch is as small as possible.

**Release branches** – Once a set of features & their associated bug fixes have been implemented and merged into the develop branch a release branch is created. The branch is assigned a name composed of the release prefix, followed by the numeric form of choice for your release.

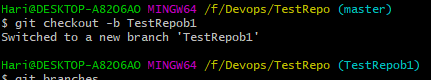
**Hotfix/Patch branches** – Hotfixes are production issues that need an immediate fix before a planned release. The development team creates a hotfix branch from master and applies the appropriate fixes.



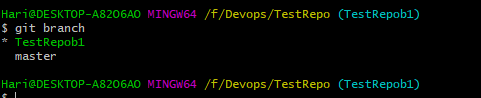
**Create a branch:**

Branch we can create local or remote please follow the below.

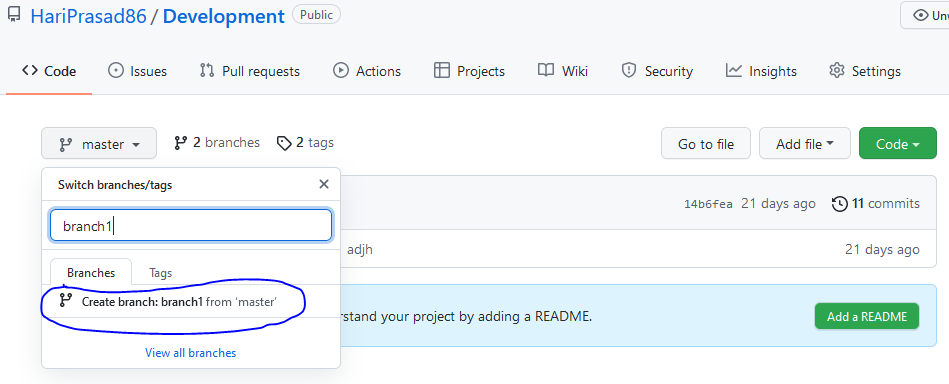
Below is the output for new branch in local repo.

****

**Command to check available branches**

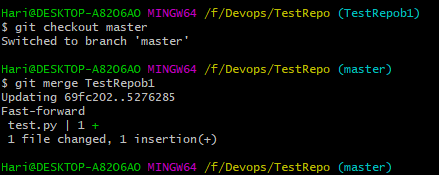
****

**Other way to create branch in github please follow below screen.**

****

**Git merge**

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together.

****

**Build Tools**

**What is a Build Tool?**

A build tool is a tool that automates everything related to building the software project. Building a software project typically includes one or more of these activities:

Generating source code (if auto-generated code is used in the project).

Generating documentation from the source code.

Compiling source code.

Packaging compiled code into JAR files or ZIP files.

Installing the packaged code on a server, in a repository or somewhere else.

**Installing Maven**

Download Maven latest Maven software from [Download latest version of Maven](https://maven.apache.org/download.cgi)

For example: **apache-maven-3.1.1-bin.zip**

Extract package

Add Maven Path in environment variable

**Maven Overview**

Maven is centered around the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

**Maven Repository**

A maven repository is a directory of packaged JAR file with pom.xml file. Maven searches for dependencies in the repositories. There are 3 types of maven repository:

Local Repository

Central Repository

Remote Repository

**Maven Local Repository**

Maven local repository is located in your local system. It is created by the maven when you run any maven command.

By default, maven local repository is %USER\_HOME%/.m2 directory. For example: C:\Users\Hari\.m2.

**Maven Central Repository**

Maven central repository is located on the web. It has been created by the apache maven community itself.

The path of central repository is: <http://repo1.maven.org/maven2/>

**Maven Remote Repository**

Maven remote repository is located on the web. Most of libraries can be missing from the central repository such as JBoss library etc, so we need to define remote repository in pom.xml file.

**Maven pom.xml file**

POM is an acronym for Project Object Model. The pom.xml file contains information of project and configuration information for the maven to build the project such as dependencies, build directory, source directory, test source directory, plugin, goals etc.

Maven reads the pom.xml file, then executes the goal.

**Elements of maven pom.xml file**

|  |
| --- |
|  |
| **project** | It is the root element of pom.xml file. |
| **modelVersion** | It is the sub element of project. It specifies the modelVersion. It should be set to 4.0.0. |
| **groupId** | It is the sub element of project. It specifies the id for the project group. |
| **artifactId** | It is the sub element of project. It specifies the id for the artifact (project). An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, and WARs. |
| **version** | It is the sub element of project. It specifies the version of the artifact under given group. |

**Example of pom.xml file below**

<project xmlns="http://maven.apache.org/POM/4.0.0"   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.javatpoint.application1</groupId>

  <artifactId>my-app</artifactId>

  <version>1</version>

 </project>

**Below command to generate Maven project**

**syntext**

**mvn archetype:generate -DgroupId={groupID} -DartifactId={artifactID} -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false**

**example below**

**mvn archetype:generate -DgroupId=devops -DartifactId=devopsProject -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false**

**command for web application and creating war package**

**mvn archetype:generate -DgroupId=com.javatpoint -DartifactId=CubeGeneratorWeb -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false**