Chapter – 2

Version Control-GIT

2	2. Version Control-GIT 2.1. Introduction to GIT 2.2. What is Git 2.3. About Version Control System and Types 2.4. Difference between CVCS and DVCS 2.5. A short history of GIT 2.6. GIT Basics 2.7. GIT Command Line 2.8. Installing Git 2.9. Installing on Linux 2.10. Installing on Windows 2.11. Initial setup 2.12. Git Essentials 2.13. Creating repository 2.14. Cloning, check-in and committing 2.15. Fetch pull and remote 2.16. Branching 2.17. Creating the Branches, switching the branches, merging 2.18. The branches.	15	3

GIT

GIT :

Git is a version control system.

GIT :

Git helps you **keep track of code changes.**

GIT :

Git is used to collaborate on code.

GIT :

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

GIT :

It is used for:

-> Tracking code changes

-> Tracking who made changes

-> Coding collaboration

What does Git do?

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes

  with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different

  parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Why Git?

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.

Working with Git

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**

Working with Git

- 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!
-

Git Install

You can download Git for free from the following website: https://www.git-scm.com/

GIT :

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

GIT :

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, **convenient staging areas**, and **multiple workflows**.

What is “Version Control” ?

What is Version Control ?

Version control is about the management of multiple versions of a project. To manage a version, each change (addition, edition, or removal) to the files in a project must be traced.

What is Version Control ?

Version control records each change made to a file (or a group of files) and offers a way to undo or roll back each change.

What is Version Control ?

With Version control, multiple people can work on their copy of the project (called branches) and only merge those changes to the main project when they (or the other than team members) are satisfied with the work.

What is Version Control ?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

What is Version Control ?

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use.

What is Version Control ?

It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.

What is Version Control ?

Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Local Version Control Systems :

One of the most popular **local VCSs** was **Source Code Control System or SCCS**, which was free bust closed source.

Local Version Control Systems :

Developed by AT&T, it was wildly used in the 1970 until Revision Control System or RCS was released.

Local Version Control Systems :

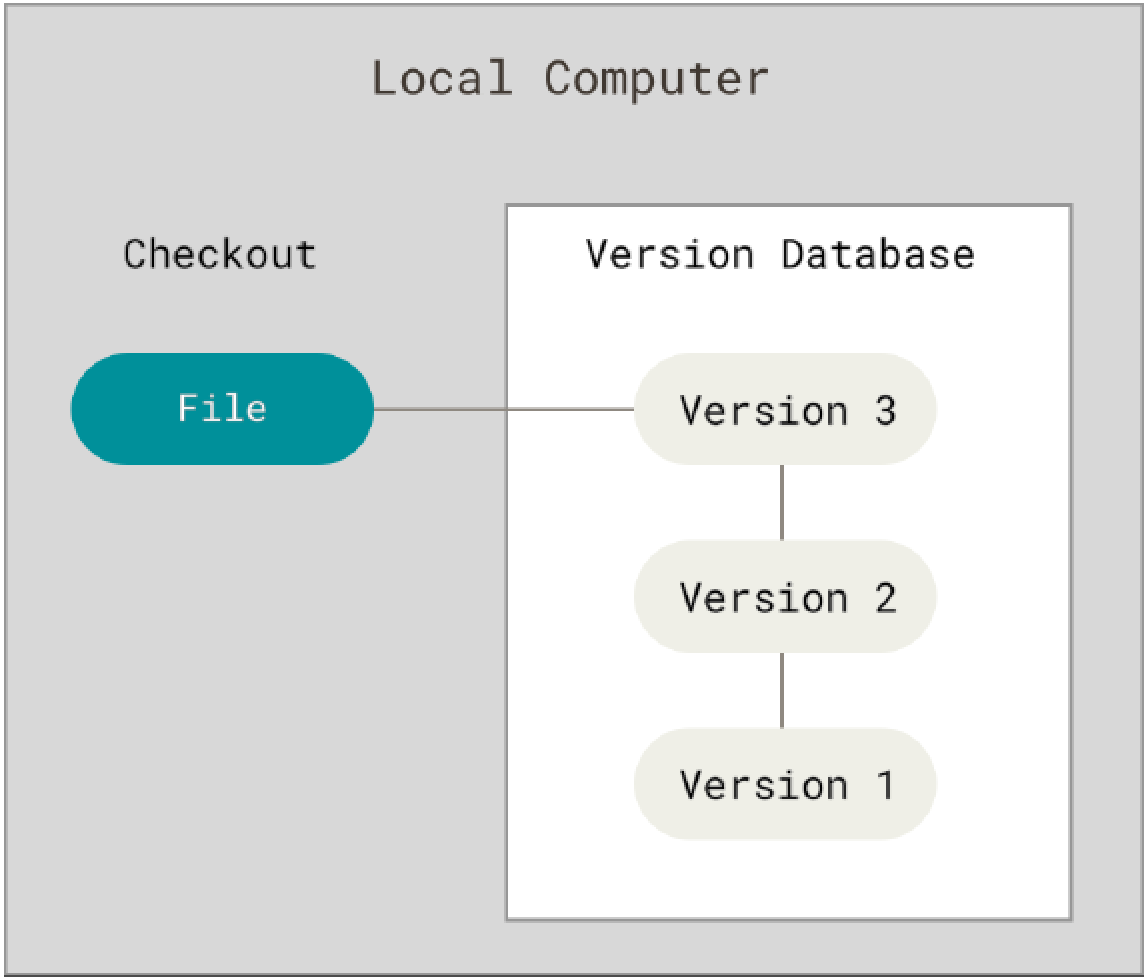
RCS became more popular than SCCS because it was **Open Source, Cross Platform, and much more effective**.

RCS is currently maintained by the **GNU Project.**

Local Version Control Systems :

One of the **drawbacks** of these two local VCSs was that they only worked on a file at a time; there was no way to track an entire project with them.

Local Version Control Systems :



Local Version Control Systems :

A **local version control system** is a local database located on your local computer, in which every file change is stored as a patch. Every patch set contains only the changes made to the file since its last version.

In order to see what the file looked like at any given moment, it is necessary to add up all the relevant patches to the file in order until that given moment.

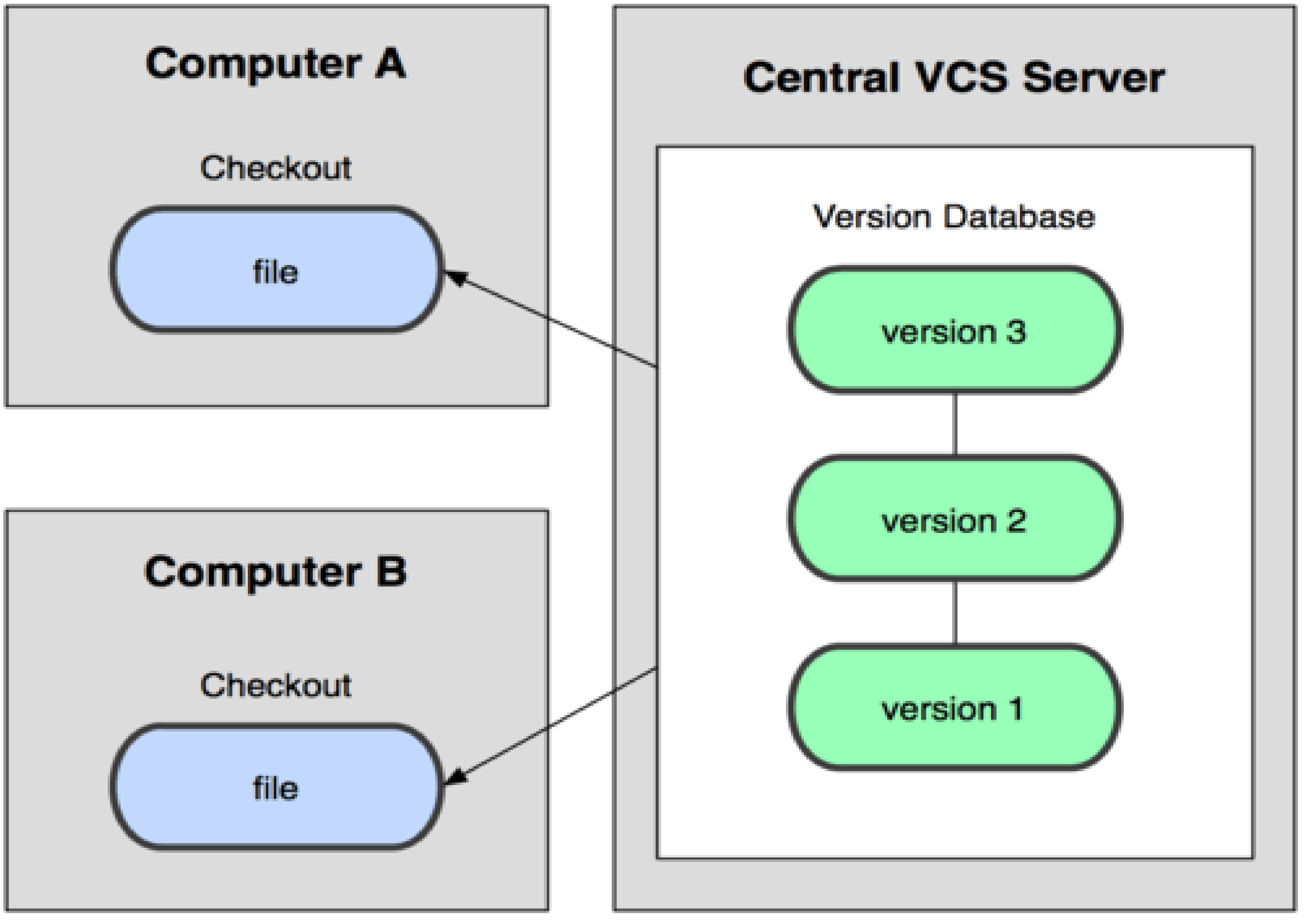
Local Version Control Systems :

The main problem with this is that everything is **stored locally**. If anything were to happen to the local database, all the patches would be lost. If anything were to happen to a single version, all the changes made after that version would be lost.

Also, collaborating with other developers or a team is very hard or nearly impossible.

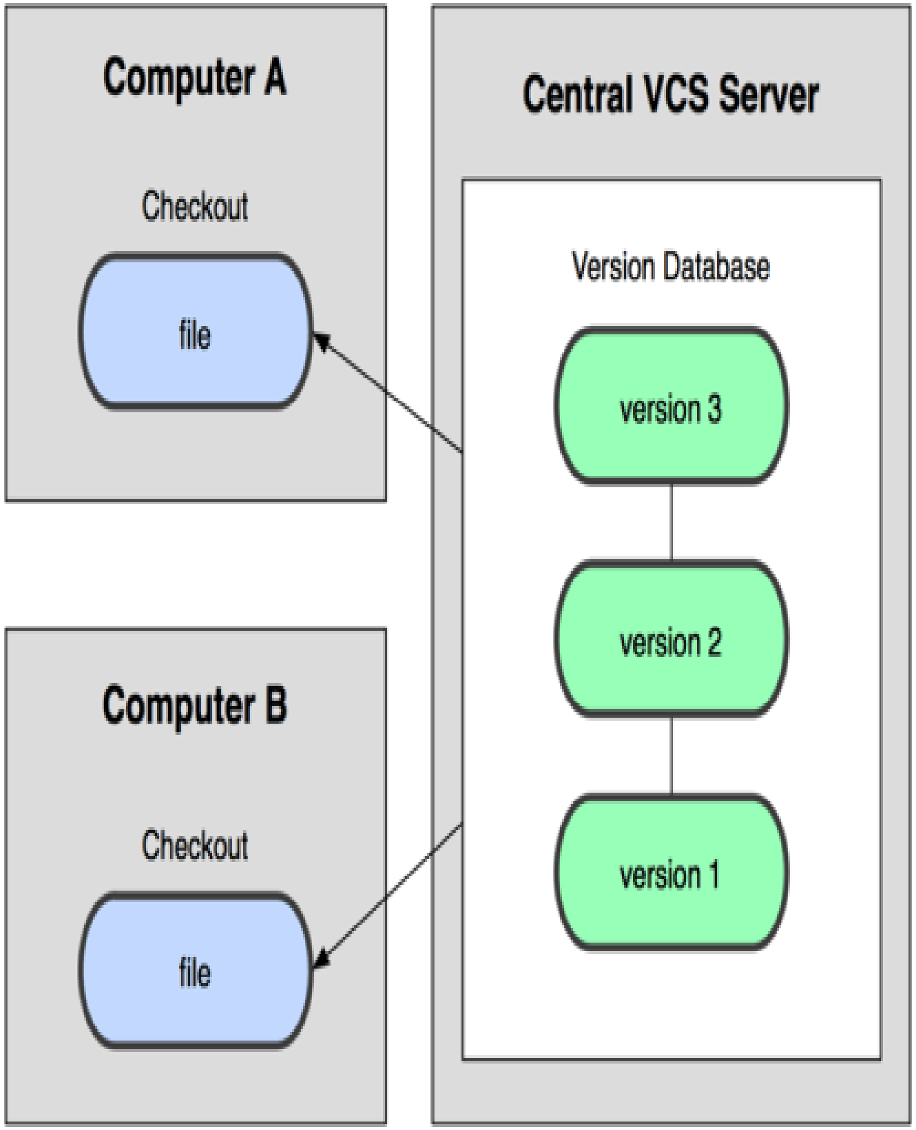
Centralized Version Control Systems

Centralized Version Control Systems



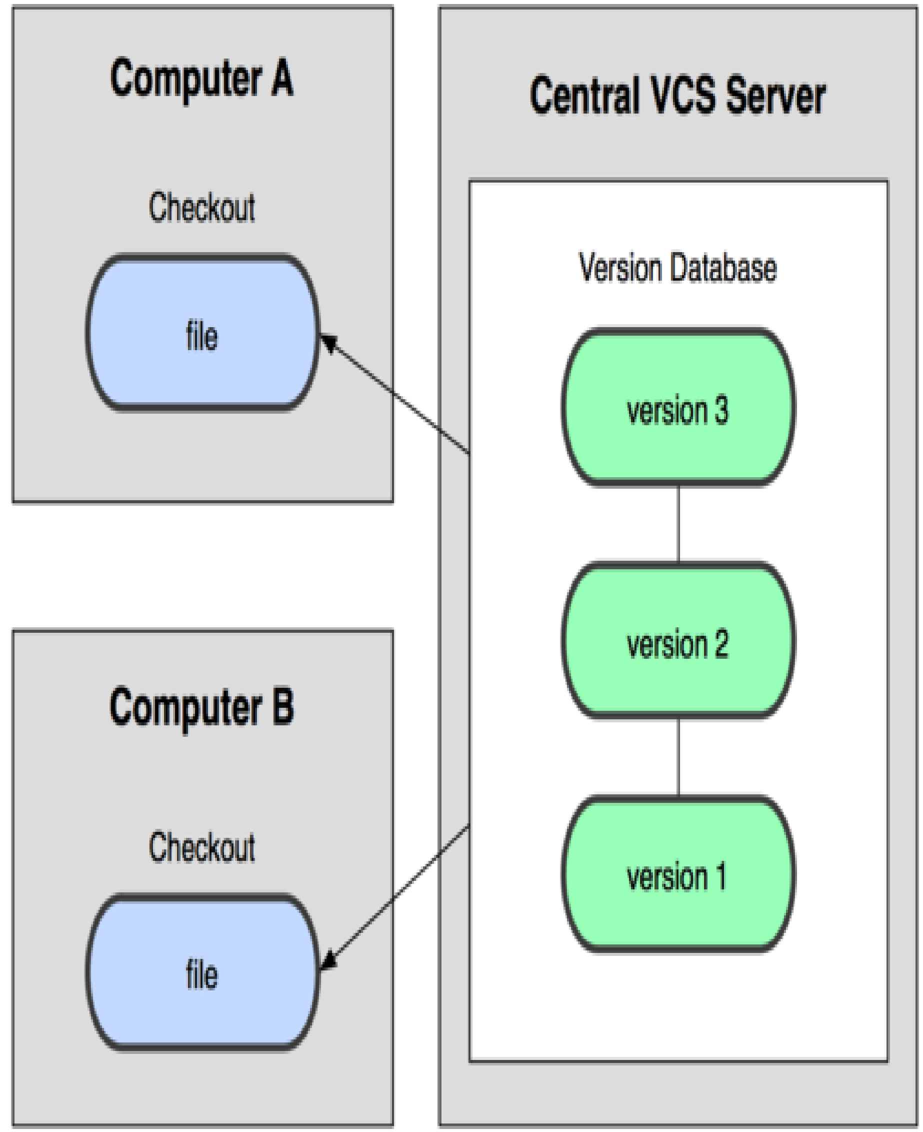
Centralized Version Control Systems

Centralized VCS as known as one of its application, subversion (SVN) or maybe other application Concurrent Versions System (CVS).

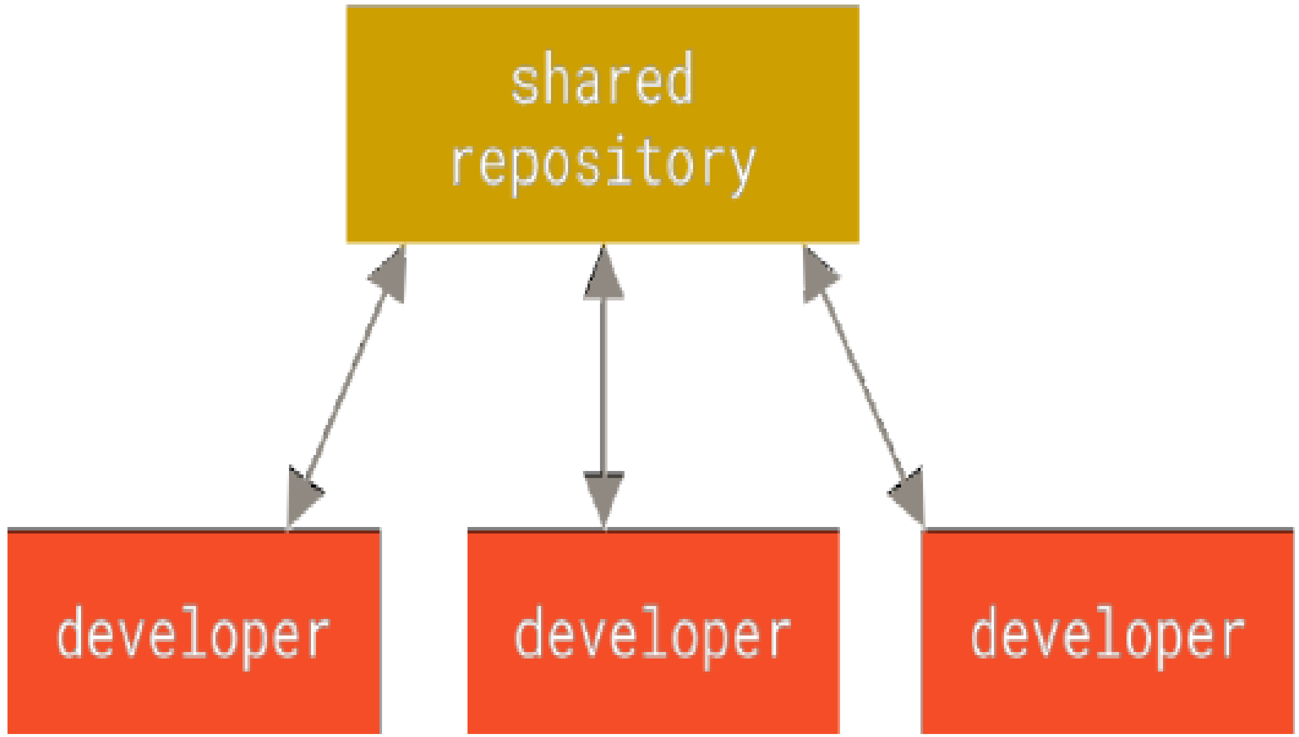


Centralized Version Control Systems

Centralized CVS is a way to track changes on project, but differently with local VCS, this way we save the snapshot of the application on the server, with history changes saved in the server.

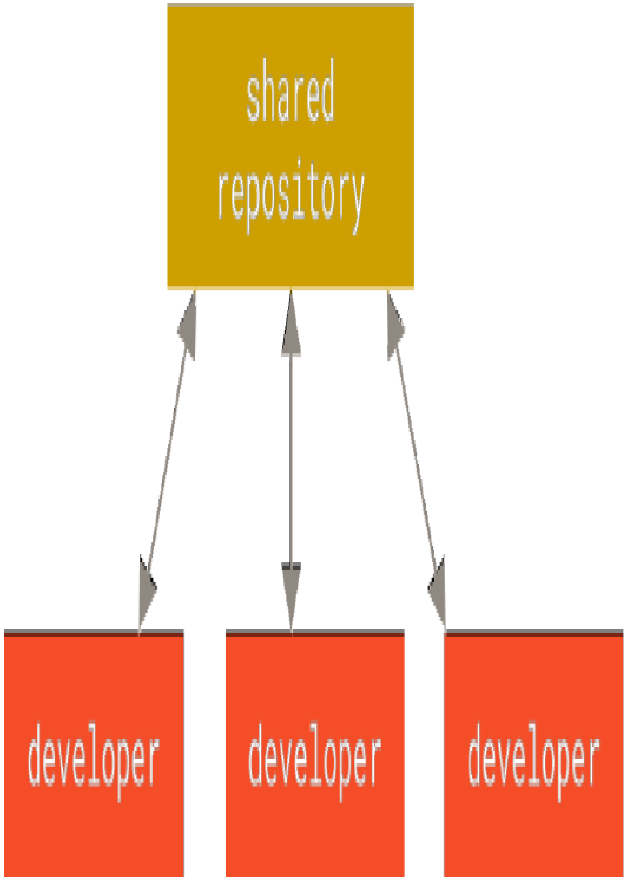


Centralized Version Control Systems :



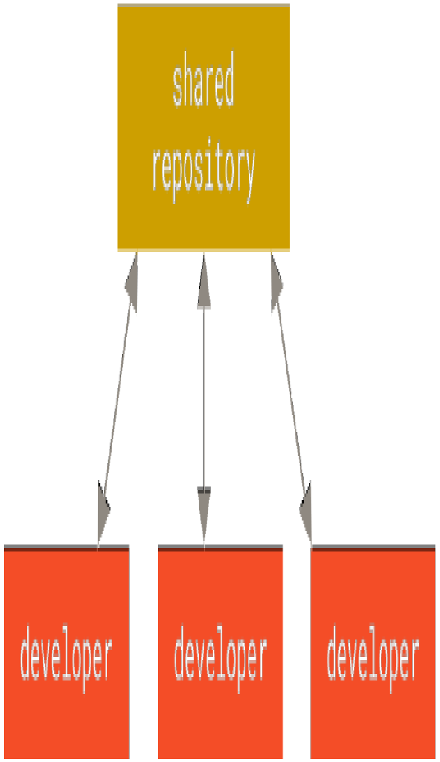
Centralized Version Control Systems :

Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.



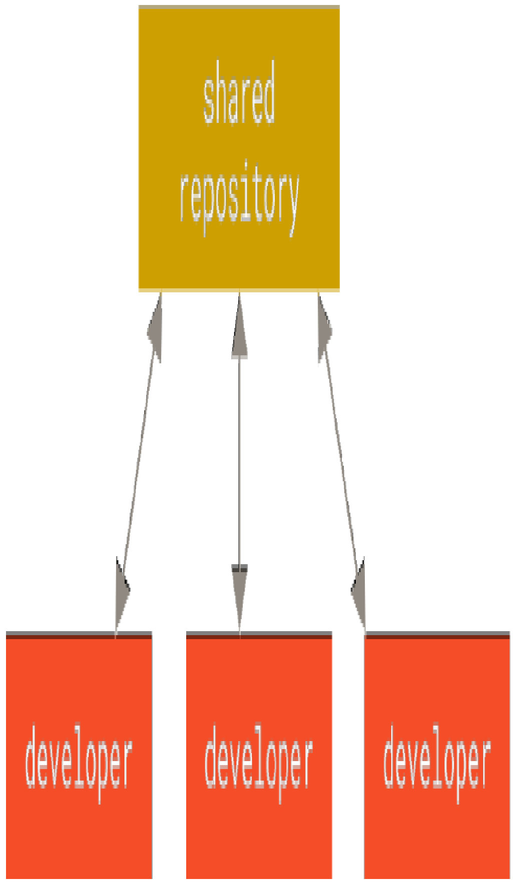
Centralized Version Control Systems :

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.



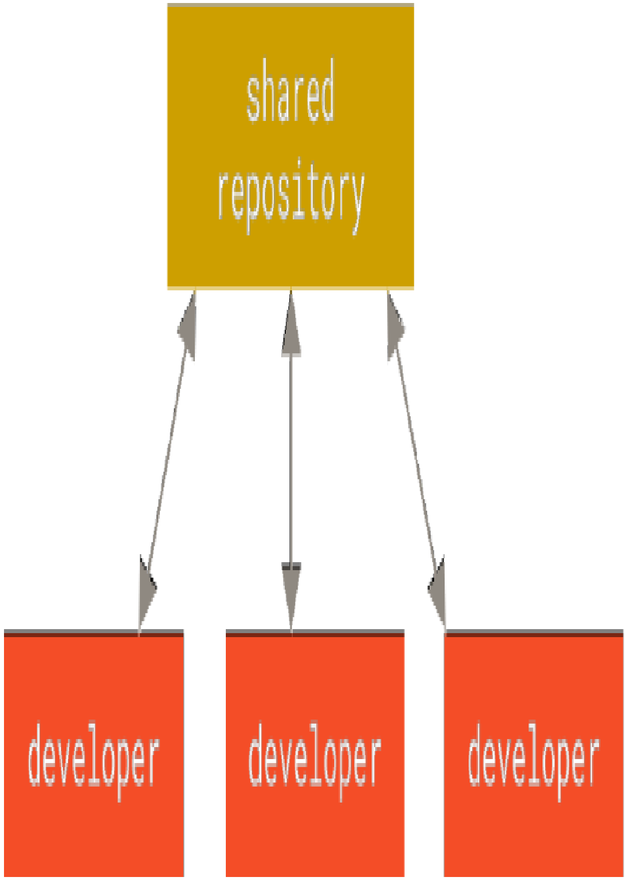
Centralized Version Control Systems :

This setup also has some **serious downsides.** The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.



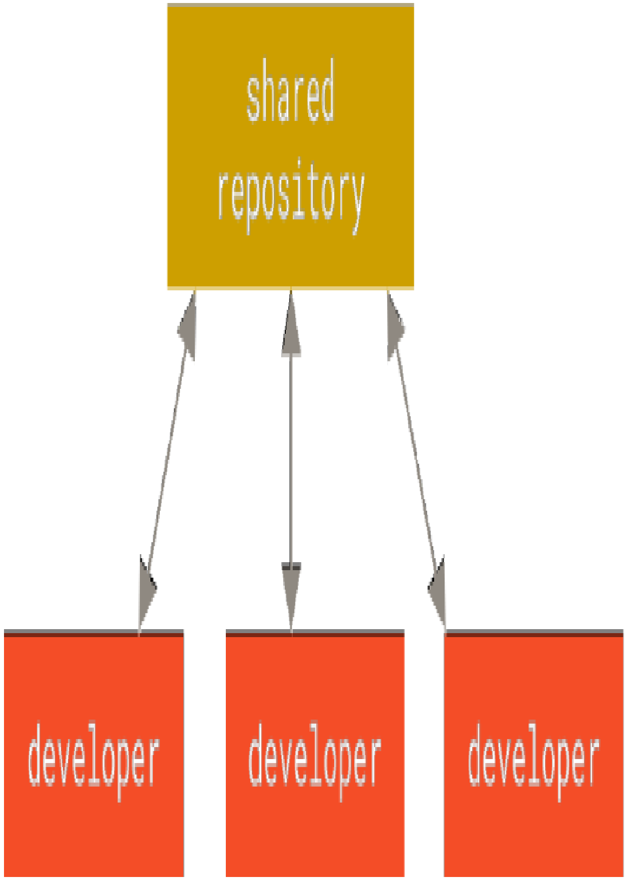
Centralized Version Control Systems :

If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines.



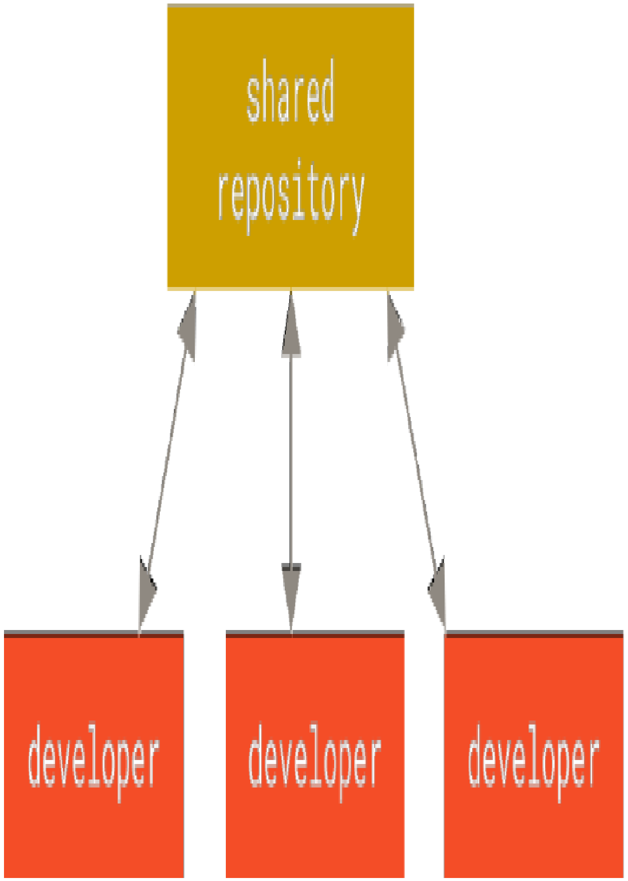
Centralized Version Control Systems :

Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.



Centralized Version Control Systems :

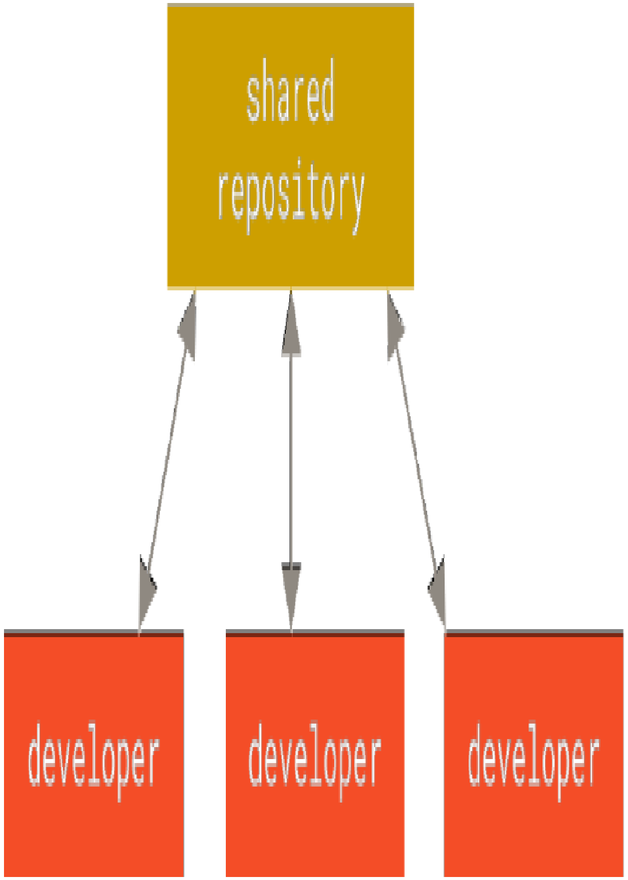
The main problem faced by team using a centralized VCS is that once a file is being used by someone, that file is locked and the other team members can't work on it.



Centralized Version Control Systems :

This creates a lot of delays in development and is generally source to a lot of frustration for contributors.

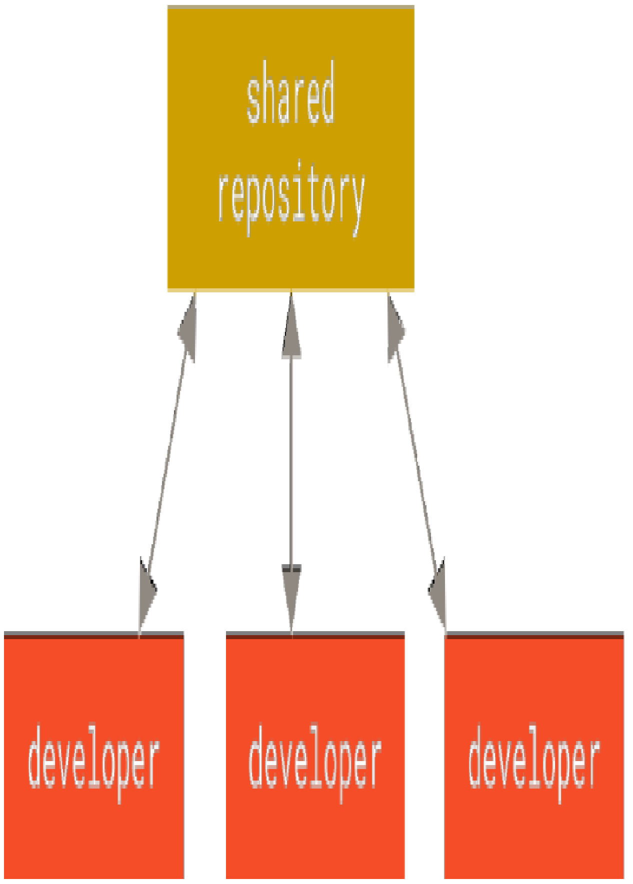
And the more members are on the team, the more problem arise.



Centralized Version Control Systems :

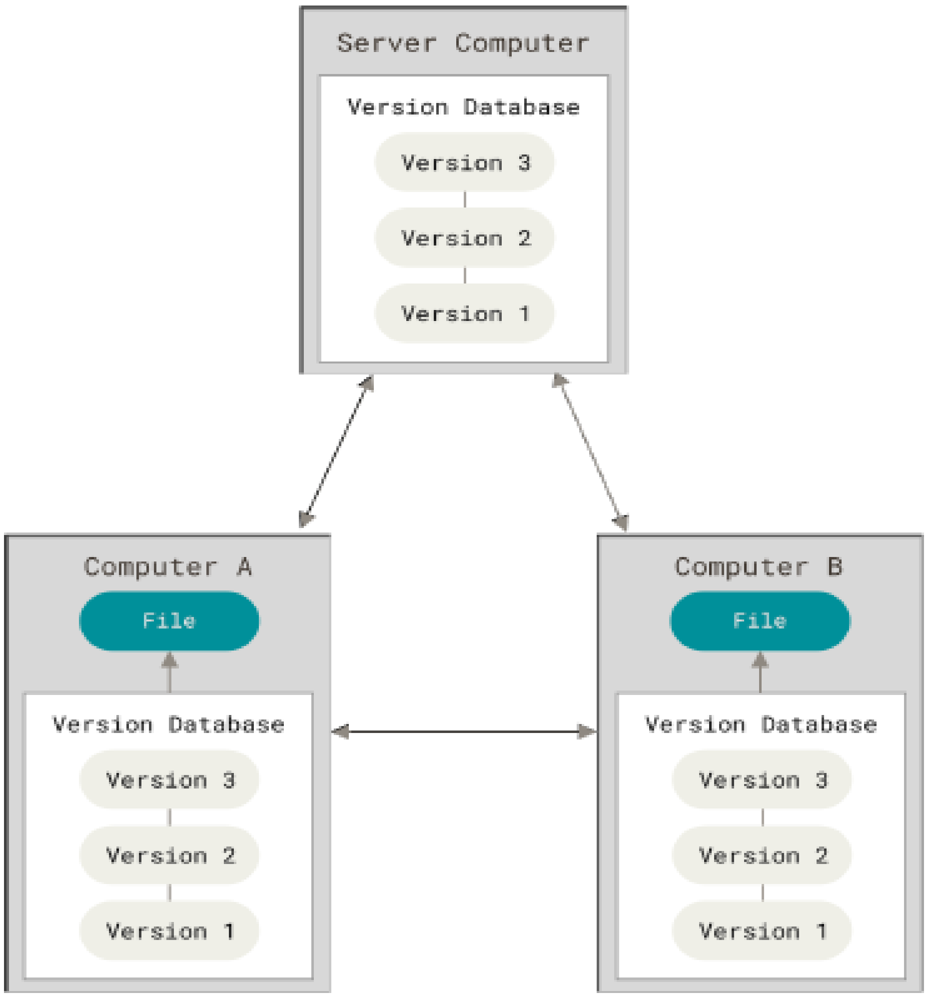
This creates a lot of delays in development and is generally source to a lot of frustration for contributors.

And the more members are on the team, the more problem arise.

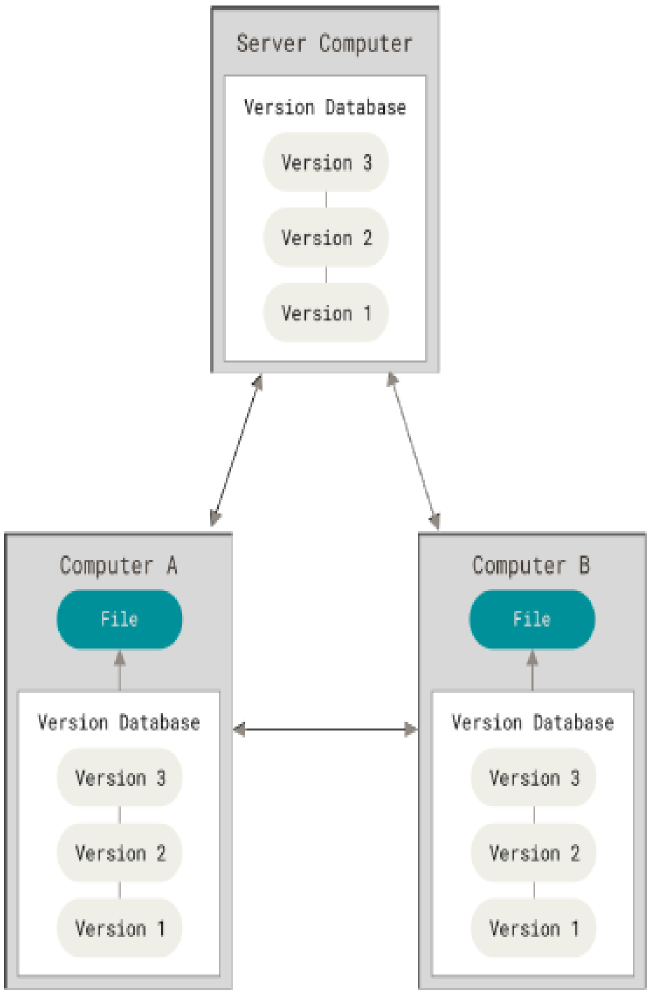


Distributed Version Control Systems

Distributed Version Control Systems

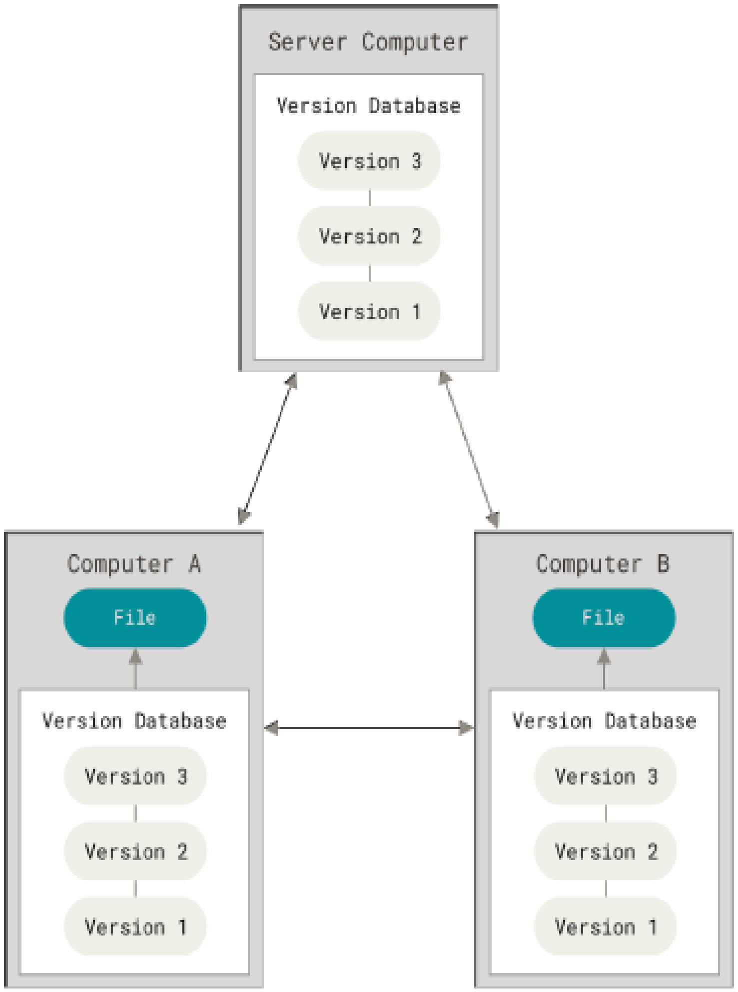


Distributed Version Control Systems



This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they **fully mirror the repository, including its full history.**

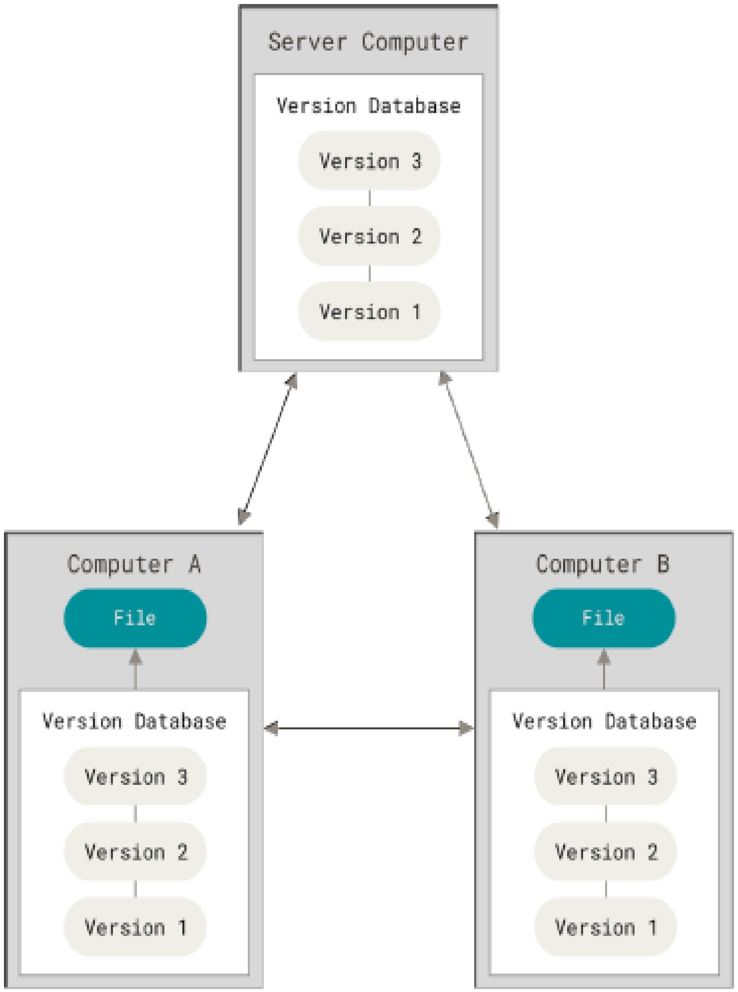
Distributed Version Control Systems



Thus, if any **server dies**, and these systems were collaborating via that server**, any of the client repositories can be copied back up to the server to restore it**.

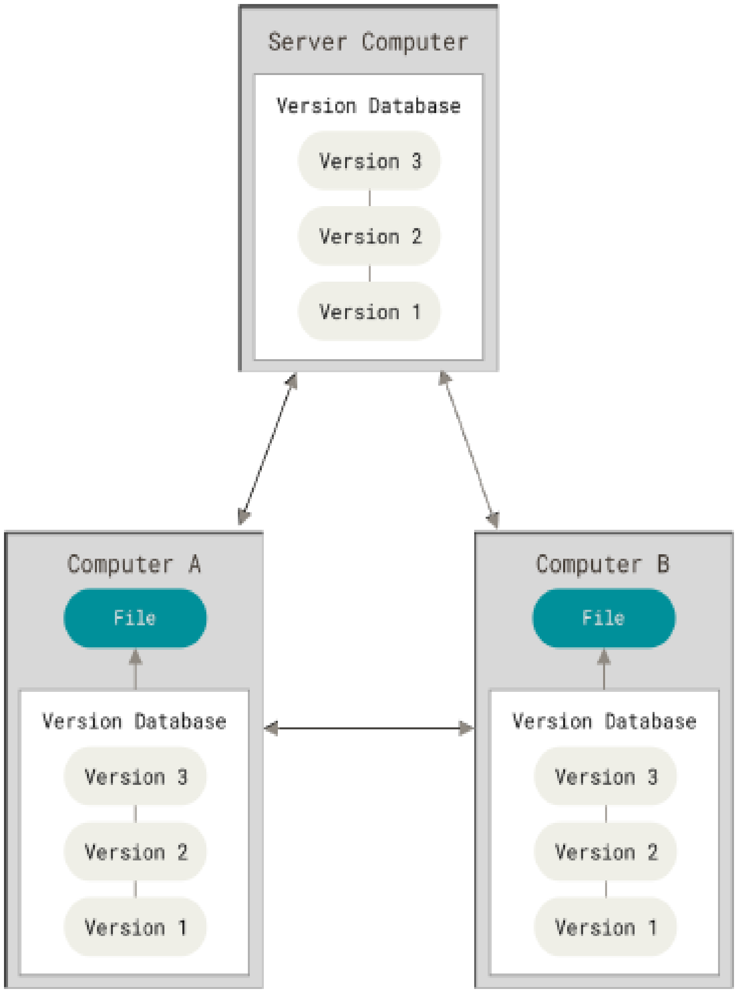
Every clone is really a full backup of all the data.

Distributed Version Control Systems



Many of these systems deal pretty well with having several remote repositories they can work with, so you can **collaborate with different groups of people in different ways simultaneously within the same project.**

Distributed Version Control Systems



This allows you **to set up several types of workflows** that aren’t possible in centralized systems, such as hierarchical models.

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

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)

Since its birth in **2005,** Git has evolved and matured to be easy to use and yet retain these initial qualities.

It’s amazingly fast, it’s very efficient with large projects, and it has an incredible branching system for non-linear development

Distributed Version Control Systems

Distributed VCS works nearly the same as centralized VCS but with a big difference :

- There is no main server that holds all the history.
- Each client has a copy of the repository (along with the change history) instead of checking out a single server.
- This greatly lowers the chance of losing everything as each client has a clone of the project.

Distributed Version Control Systems

Distributed VCS works nearly the same as centralized VCS but with a big difference :

- With a distributed VCS, the concept of having a “main server” gets blurred because each client essentially has all the power within their own repository.

Distributed Version Control Systems

This greatly encouraged the concept of “forking” within the Open Source Community.

Forking is the act of cloning a repository to make your own changes and have a different take on the project. The main benefit of forking is that you could also pull changes from other repositoies if you see fit (and others can do the same with your changes).

Distributed Version Control Systems

A **Distributed Version Control** System is generally faster than the other types of VCS because it doesn't need a network access to a remote server.

Distributed Version Control Systems

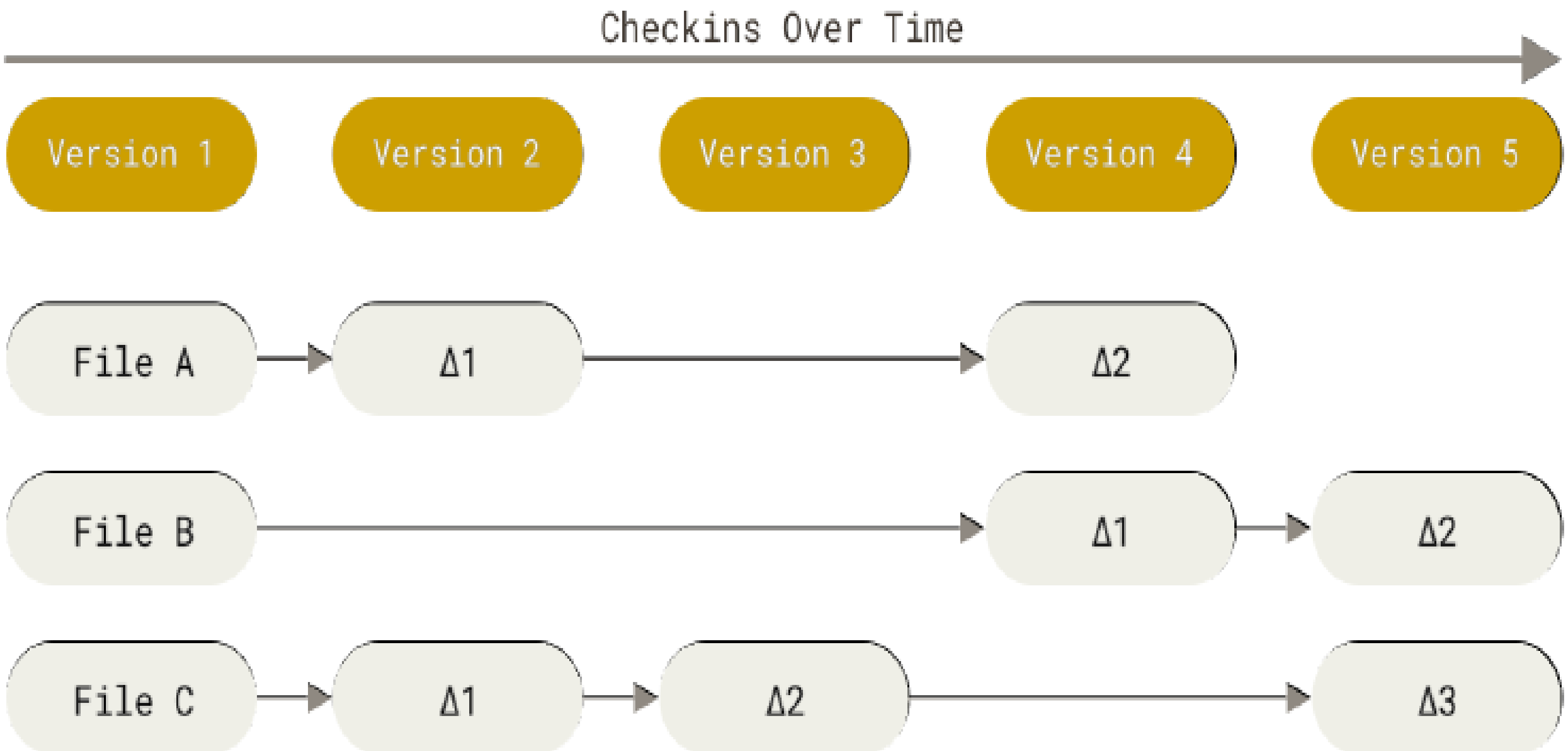
Nearly everything is done locally. There is also a slight difference with how it works : instead of tracking the changes between versions, it tracks all changes as "patches". This means that those patches can be freely exchanged between repositories, so there is no "main" repository to keep up with.

What is Git?

What is Git? Snapshots, Not Differences :

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as ***delta-based*** **version control**).

What is Git? Snapshots, Not Differences :



Storing data as changes to a base version of each file

What is Git? Snapshots, Not Differences :

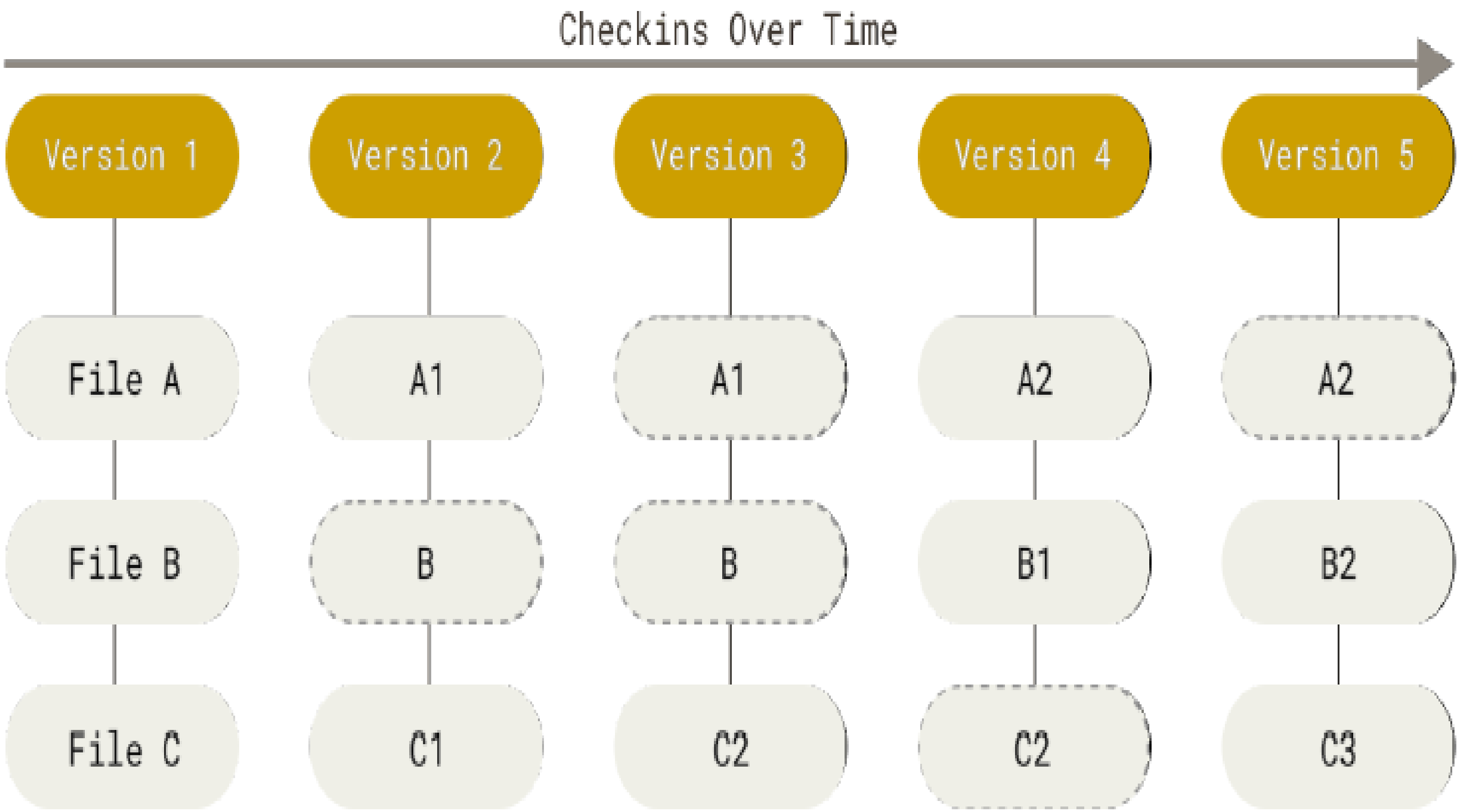
Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem.

With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

What is Git? Snapshots, Not Differences :

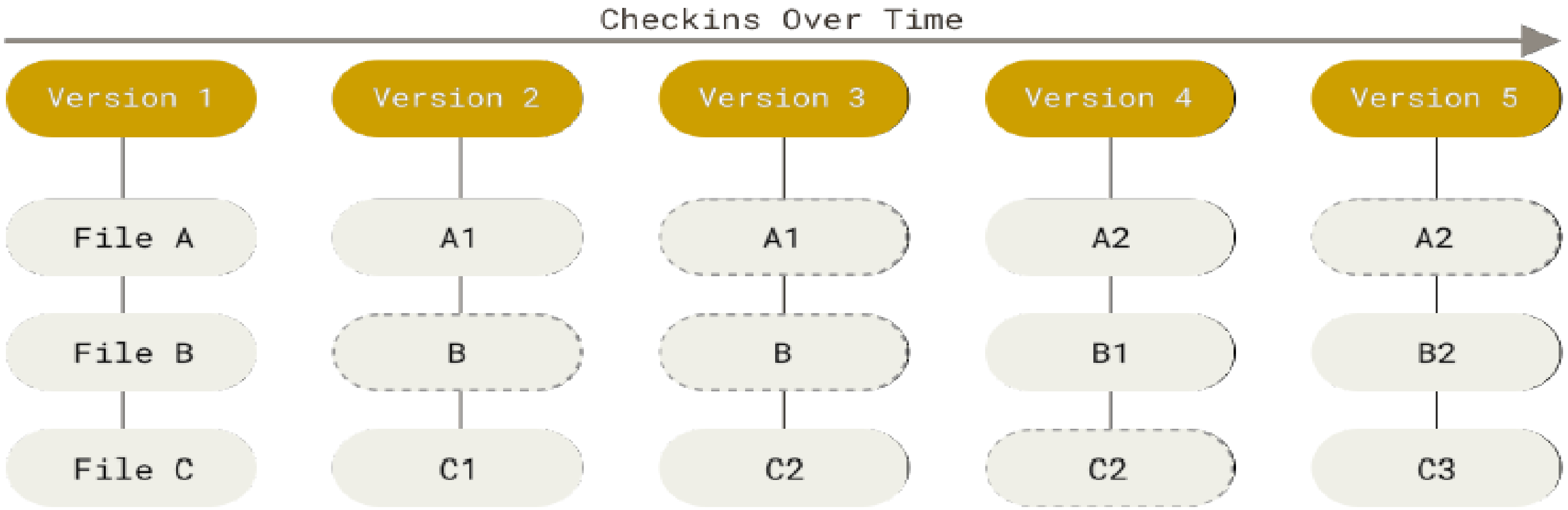
To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

What is Git? Snapshots, Not Differences :



Storing data as snapshots of the project over time

What is Git? Snapshots, Not Differences :



Storing data as snapshots of the project over time

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini file system with some incredibly powerful tools built on top of it, rather than simply a VCS.

What is Git? Every Operation is Locals :

Most operations in Git need only local files and resources to operate — **generally no information is needed from another computer on your network**.

If you're used to a CVCS where most operations have that network latency overhead.

Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

What is Git? Every Operation is Locals :

For example, to browse the history of the project, **Git doesn’t need to go out to the server to get the history and display it for you** — **it simply reads it directly from your local database.**

What is Git? Git Has Integrity :

Everything in Git is checksummed before it is stored and is then referred to by that **checksum.**

This means it’s impossible to change the contents of any file or directory without Git knowing about it.

This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it.

What is Git? Git Has Integrity :

The mechanism that Git uses for this checksumming is called a **SHA-1 hash**. This is a **40-character** string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

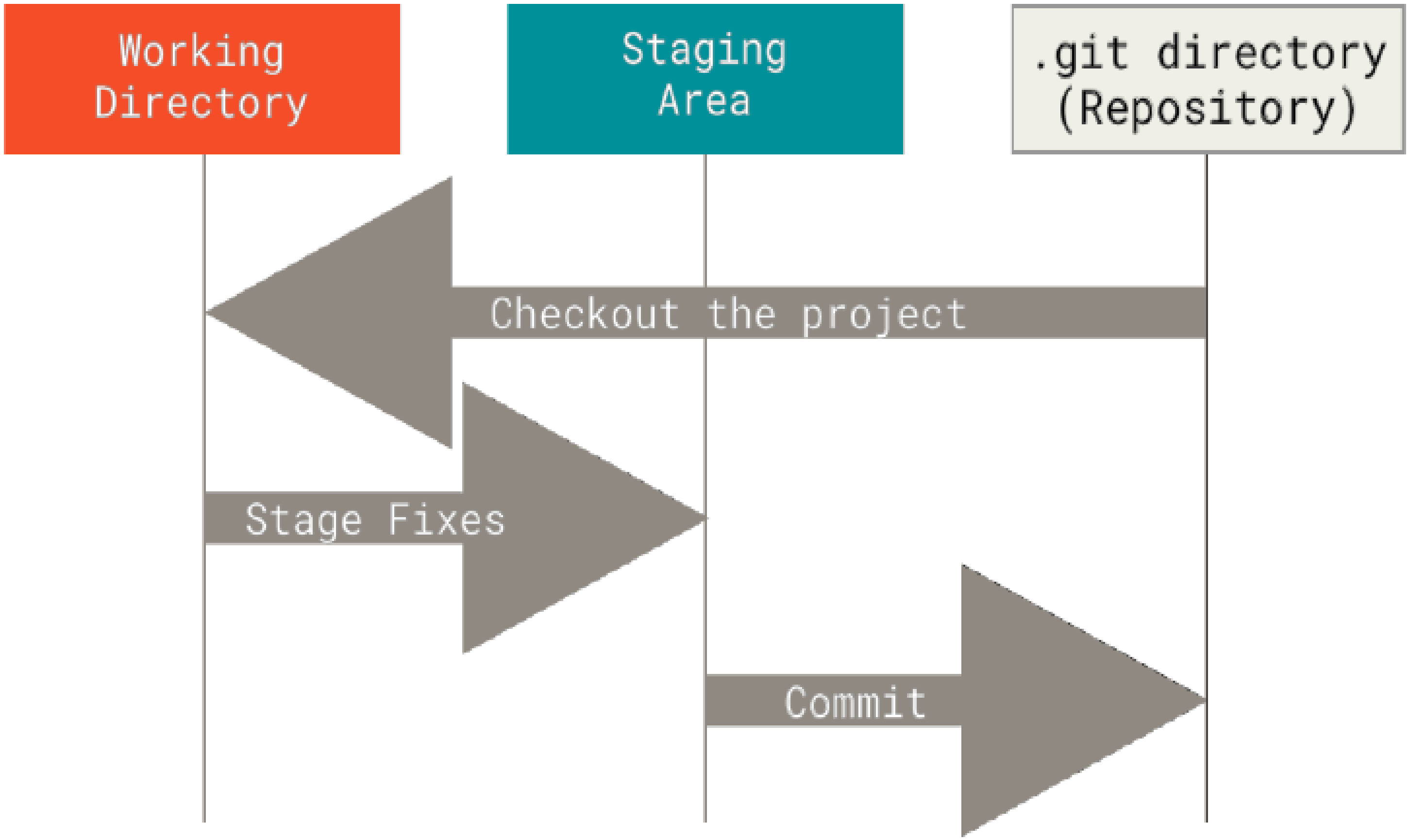
Git stores everything in its database not by file name but by the hash value of its contents.

What is Git? Git Generally Only Adds Data :

When you do actions in Git, nearly all of them only *add* data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.
As with any VCS, you can lose or mess up changes you haven’t committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

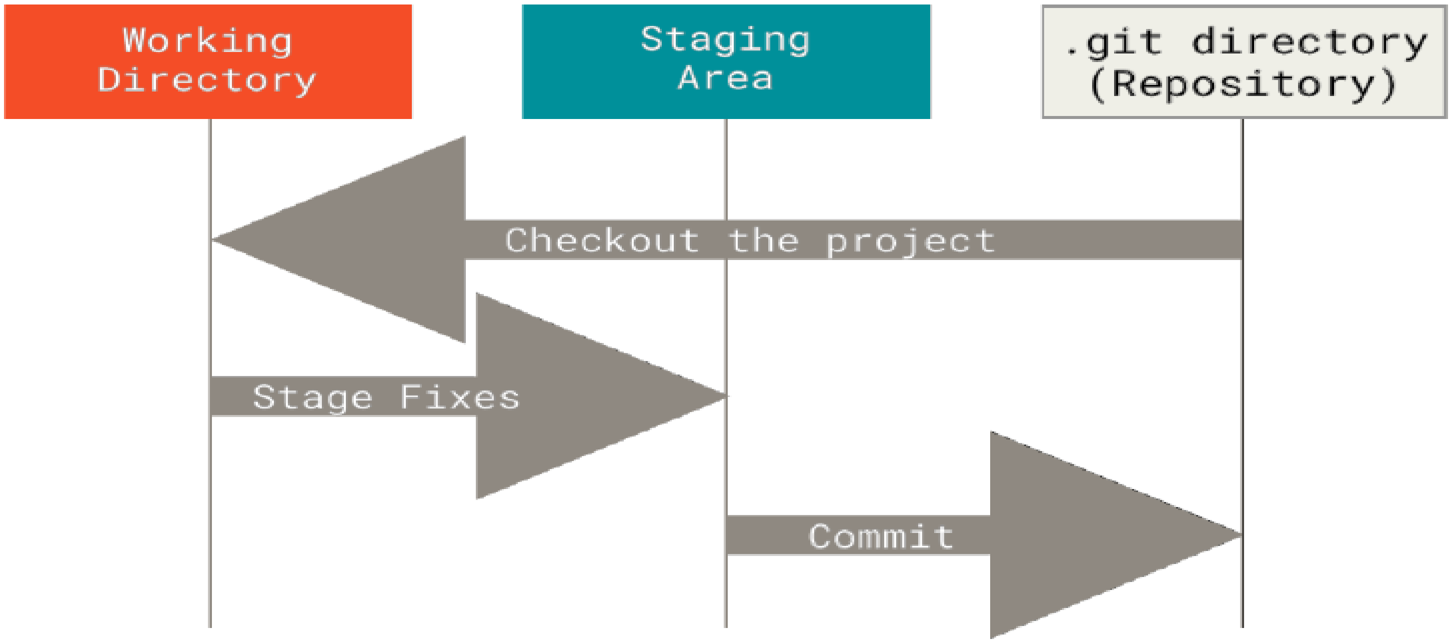
This makes using Git a joy because we know we can experiment without the danger of severely screwing things up.

What is Git? The Three States:



Working tree, staging area, and Git directory

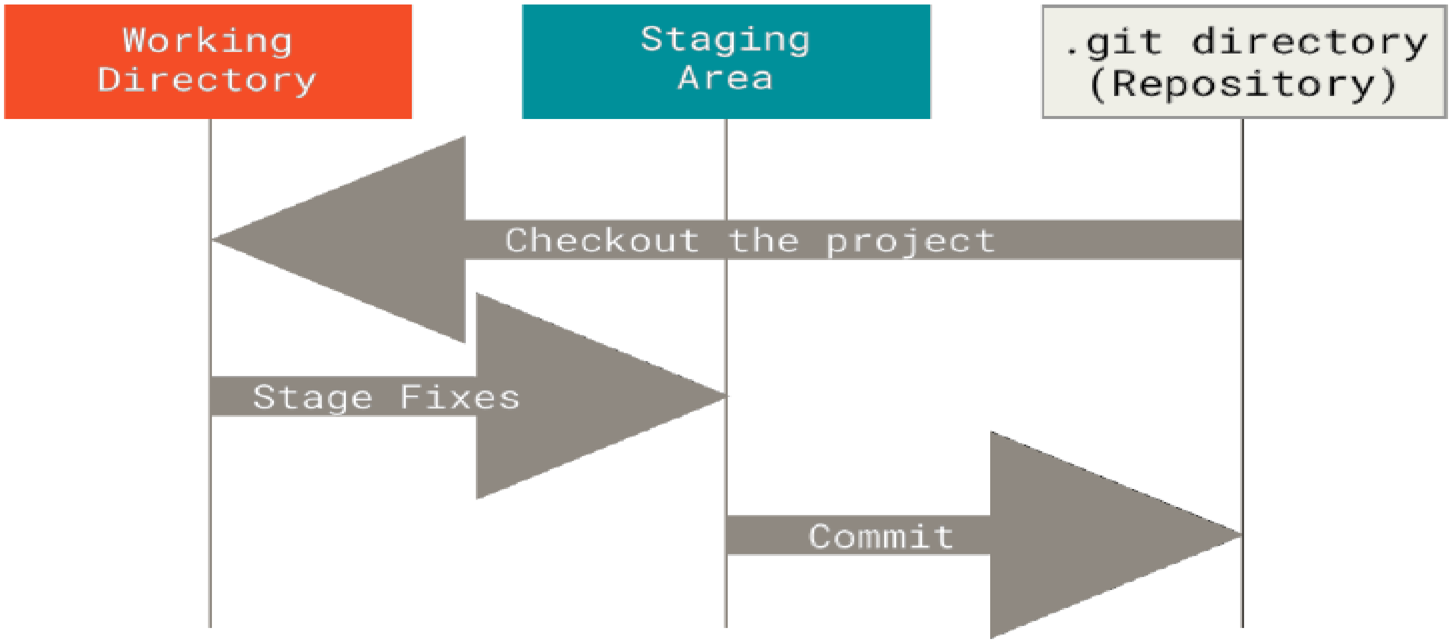
What is Git? The Three States:



Working tree, staging area, and Git directory

Git has three main states that your files can reside in: ***modified, staged,*** **and** ***committed*:**

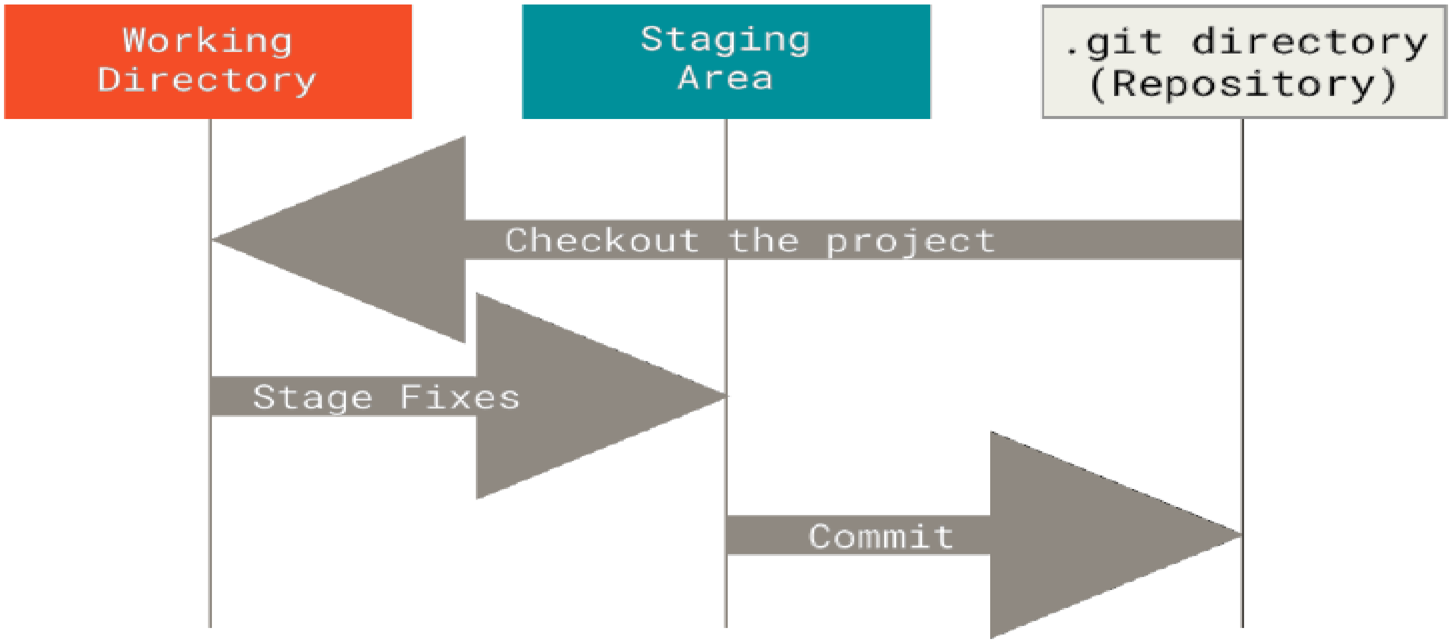
What is Git? The Three States:



Working tree, staging area, and Git directory

Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

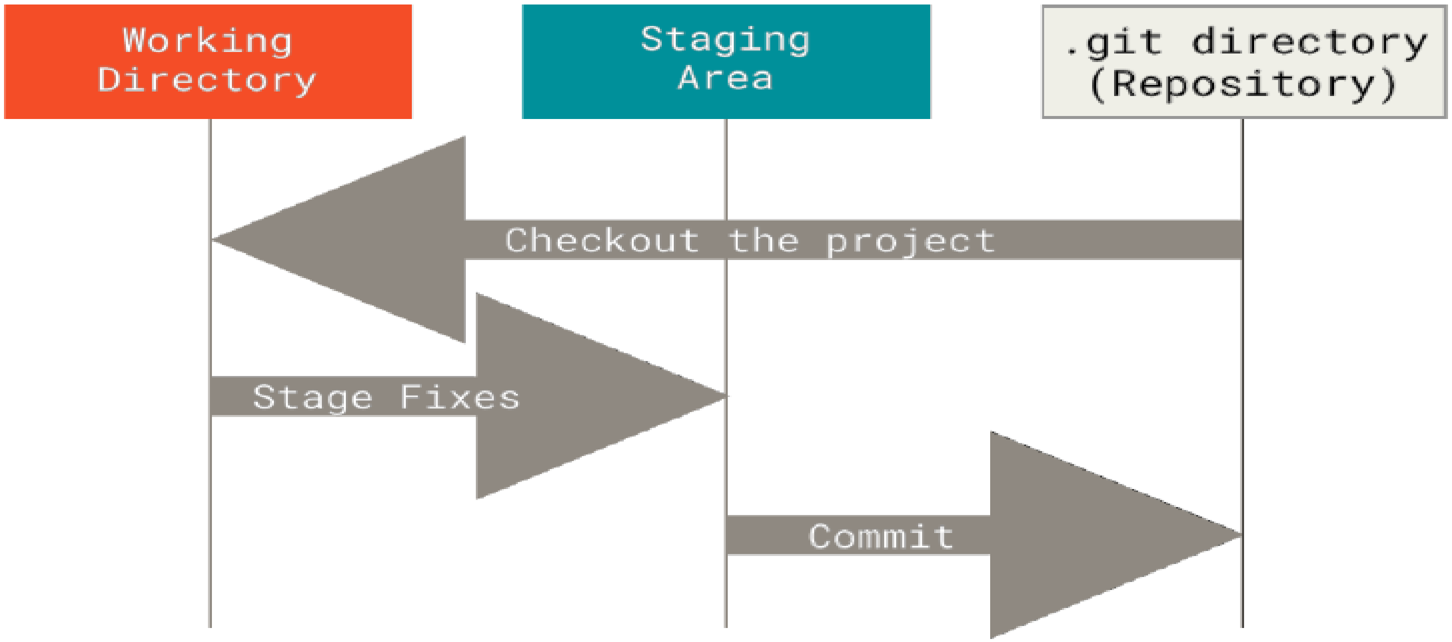
What is Git? The Three States:



Working tree, staging area, and Git directory

Committed means that the data is safely stored in your local database.

What is Git? The Three States:

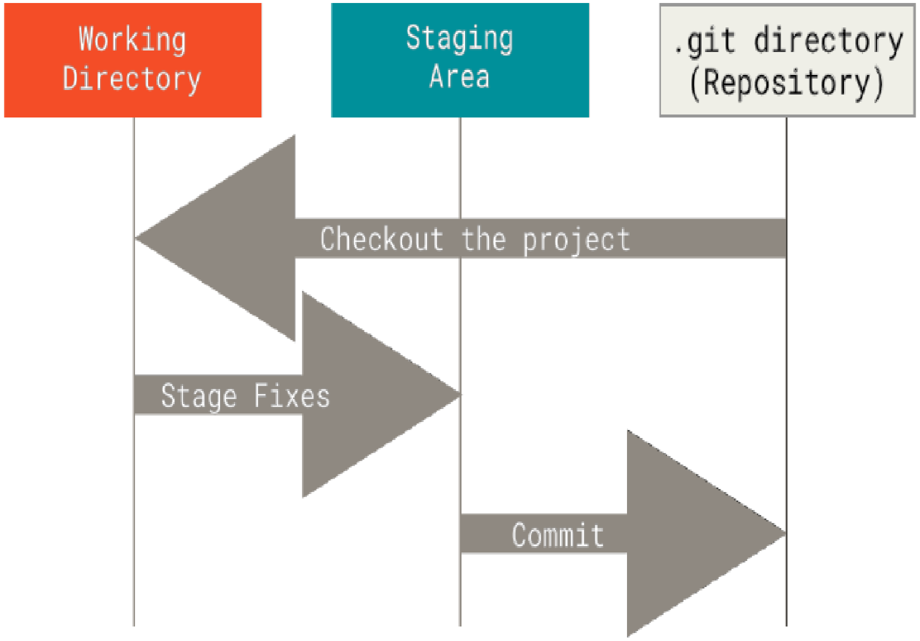


Working tree, staging area, and Git directory

This leads us to the three main sections of a Git project: the **working tree**, the **staging area**, and the **Git directory**.

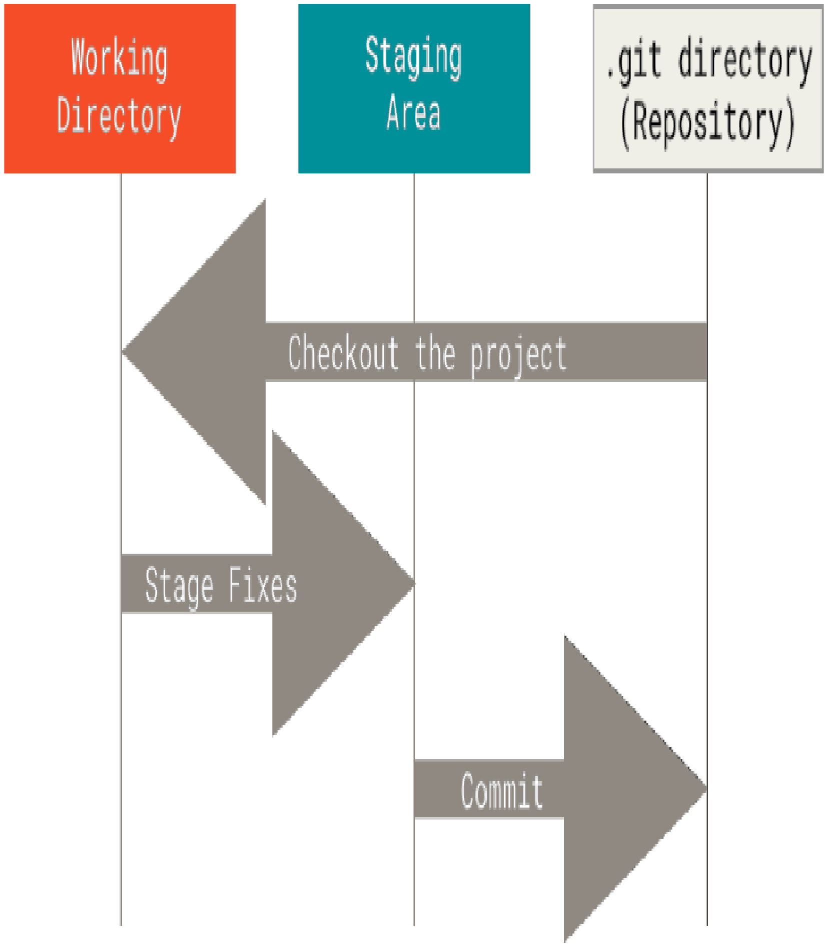
What is Git? The Three States:

The **working tree** is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.



Working tree, staging area, and Git directory

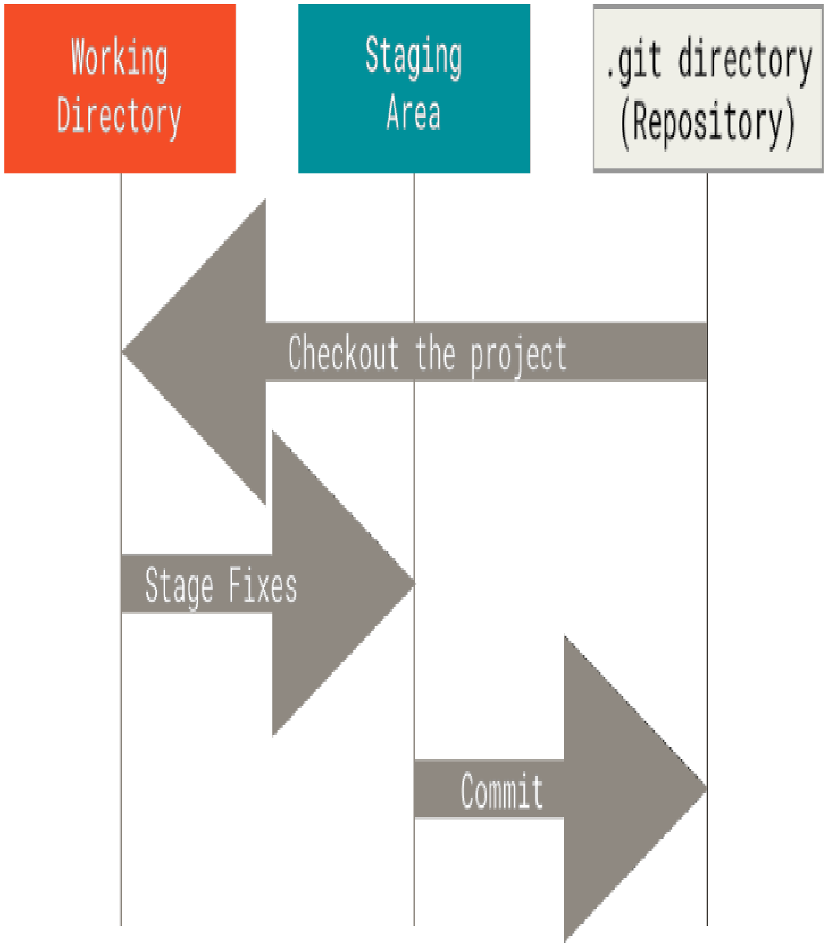
What is Git? The Three States:



Working tree, staging area, and Git directory

The **staging area** is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

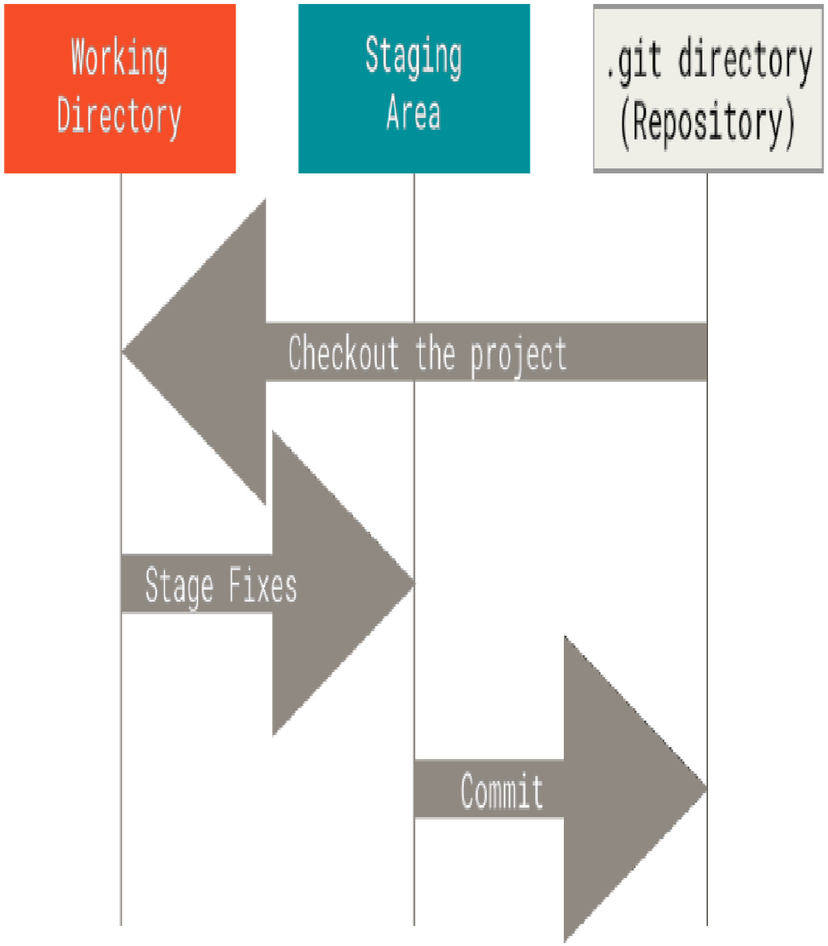
What is Git? **The Three States:**



Working tree, staging area, and Git directory

The **Git directory** is where Git stores the **metadata and object** database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

What is Git? The Three States:

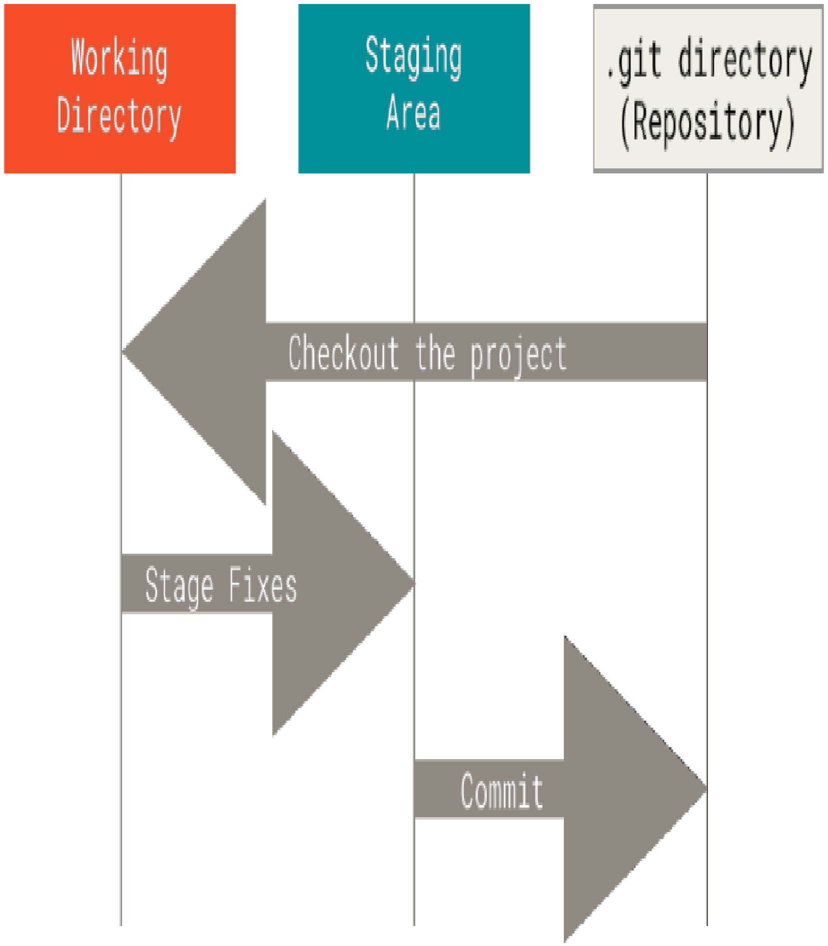


Working tree, staging area, and Git directory

The basic Git workflow goes something like this:

1. You modify files in your working tree.

What is Git? The Three States:

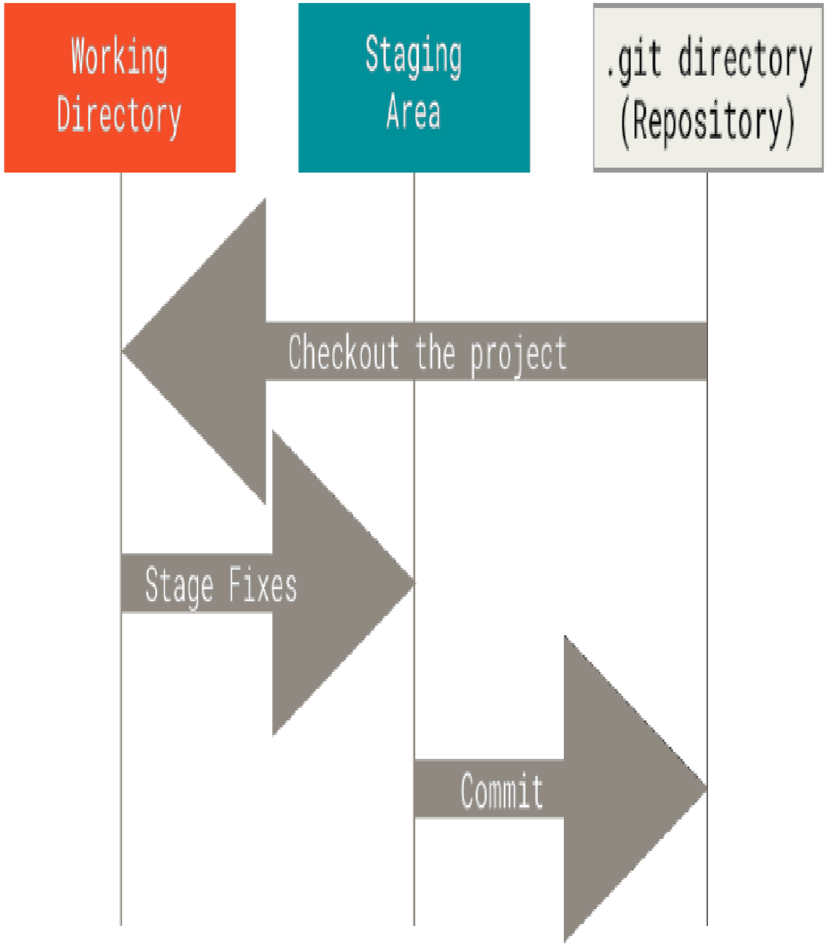


Working tree, staging area, and Git directory

The basic Git workflow goes something like this:

2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.

What is Git? The Three States:



Working tree, staging area, and Git directory

The basic Git workflow goes something like this:

3. **You do a commit**, **which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.**

If a particular version of a file is in the Git directory, it's considered ***committed***. If it has been modified and was added to the staging area, it is ***staged*****. And if it was changed since it was checked out but has not been staged, it is *modified*.**

There are a lot of different ways to use Git. There are the original **1] command-line tools**, and there are many **2] graphical user interfaces** of varying capabilities.

- command-line

What is Git ?

Git is a distributed version control system, but it is faster and works better with large projects.

It is faster and works better with large projects.

What can Git do?

It works great with tracking changes. You can

- **Go back and forth between versions**
- **Review the differences between those versions**
- **Check the change history of a file**
- **Tag a specific version for quick referencing**

What can Git do?

Git is also a great tool for teamwork. You can

- Exchange “changesets” between repositories

- Review the changes made by others

What can Git do?

One of the main features of Git is its Branching System. A branch is a copy of a project which you can work on without messing with the repository.

This concept has been around for some time, but with Git, it is way faster and more efficient.

What can Git do?

Branching also comes along with Merging, which is the act of copying the changesets done in a branch back to the source.

Generally, you create a branch to create or test a new feature and merge that branch back when you are satisfied with the work.

How does Git work ?

Git works with snapshots, not differences. This means that it does not track the difference between two versions of a file, but takes a picture of the current state of the project.

This is why Git is very fast compared to other distributed VCS. It is also why switching between versions and branches is so fast and easy.

How does Git work ?

Git is distributed VCS, every user has their own fully fledged repository with their own history and change sets. Thus, everything is done locally except the sharing of patches or change sets.

Git takes a snapshot, it performs a checksum on it, so it knows which files were changed by comparing the checksums.

How does Git work ?

This is why Git can track changes between files and directories easily, and it also checks for any file corruption.

How does Git work ?

The main feature of Git is its “Three States” system. The states are the working directory, the staging area, and the Git directory :

-> The working directory is just the current snapshot that you are working on.

-> The staging area is where modified files are marked in their current version, ready to be stored in the database.

-> The git directory is the database where the history is stored.

How does Git work ?

Git works as follows :

- **You modify the files,**
- **Add each file you want to include in the snapshot to the staging area (git add command)**
- **Then take the snapshot and add them to the database (git commit command)**

How does Git work ?

For the terminology, we call a modified file added to the staging area “staged” and a file added to the database “committed”, So, a file goes from “modified” to “staged” to “committed”.

How does Git work ?

New branch is created, you can begin to modify the files. Git will track all the changes via checksums

Now that you made the necessary changes, it is time to put them on the staging area.

The staging area is where you put modified codes that are ready to be snapshotted.

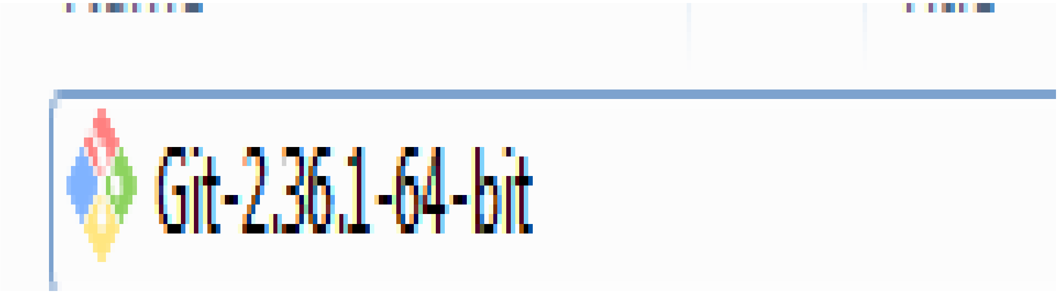
Installing Git


git --distributed-even-if-your-workflow-isnt
Search entire site...
About
Documentation
Downloads
GUI Clients
Logos
Community
The entire Pro Git book
Downloads
macOS
Windows
Linux/Unix
Older releases are available and the Git source repository is on GitHub.
Latest source Release
2.36.1
Release Notes (2022-05-05)
Download for Windows

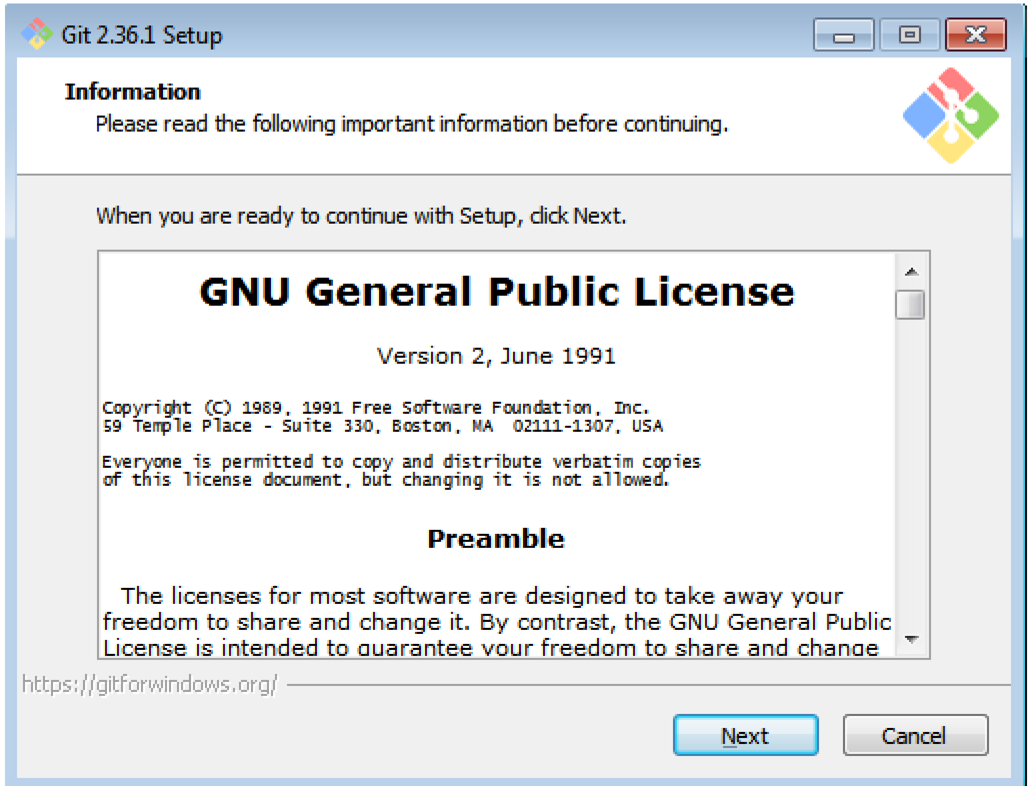
Other Git for Windows downloads

Standalone Installer

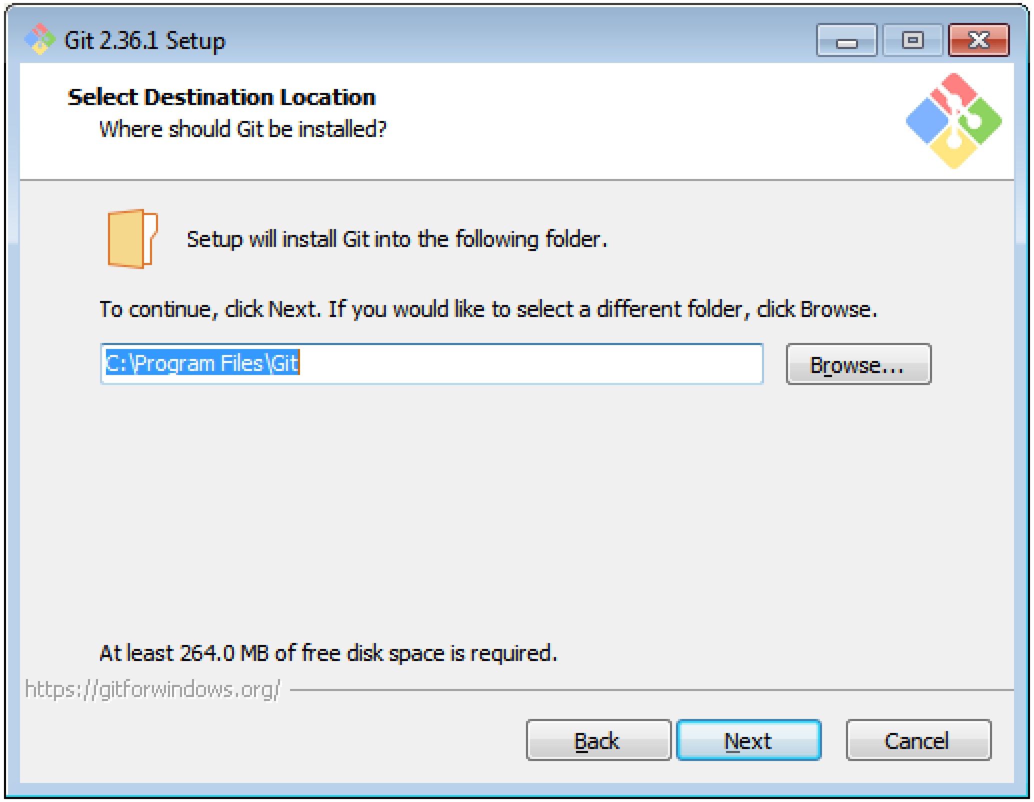
64-bit Git for Windows Setup.



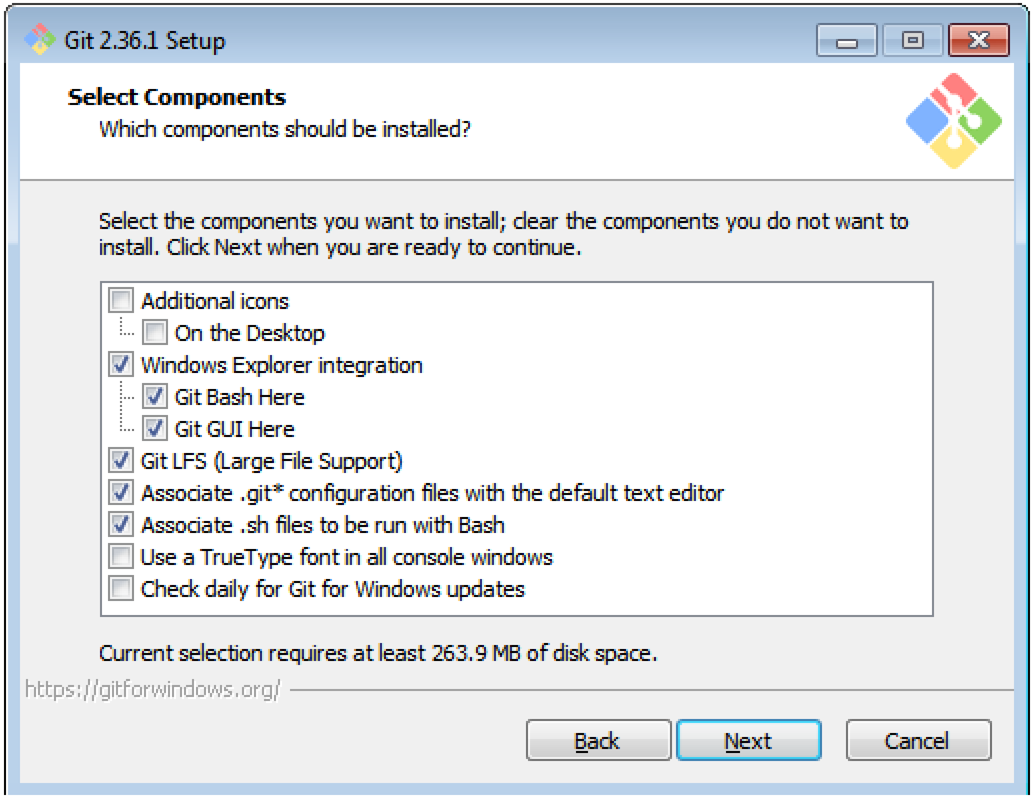
Double Click and start Installing



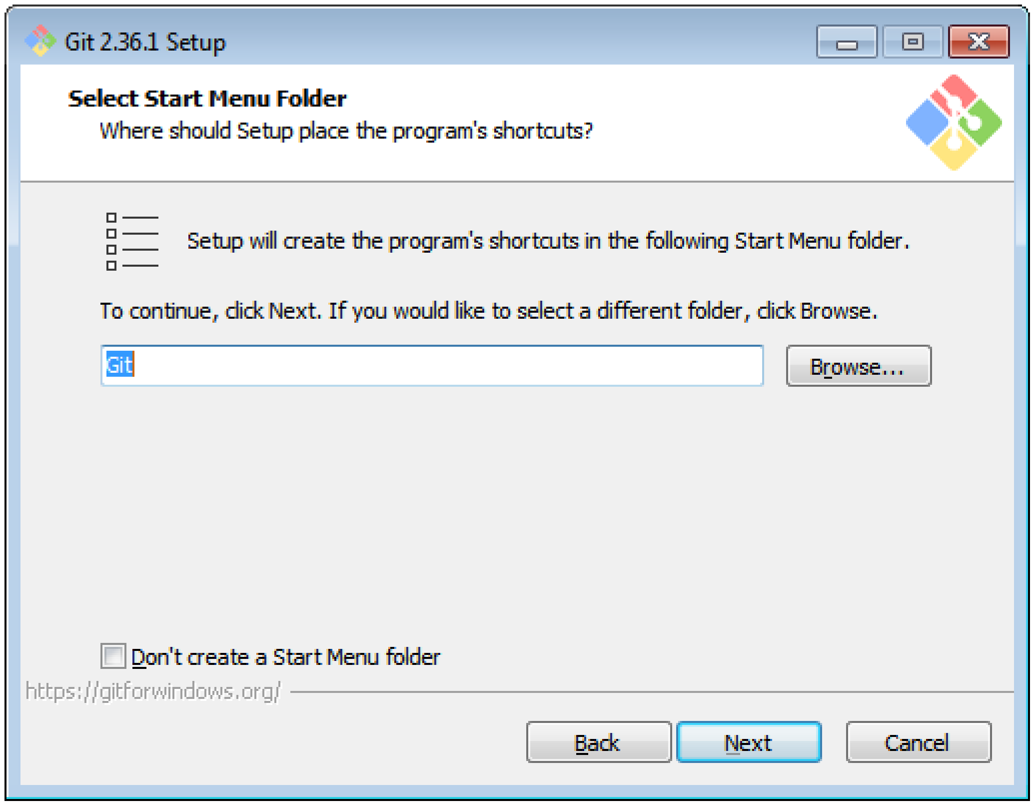
Click on Next Button



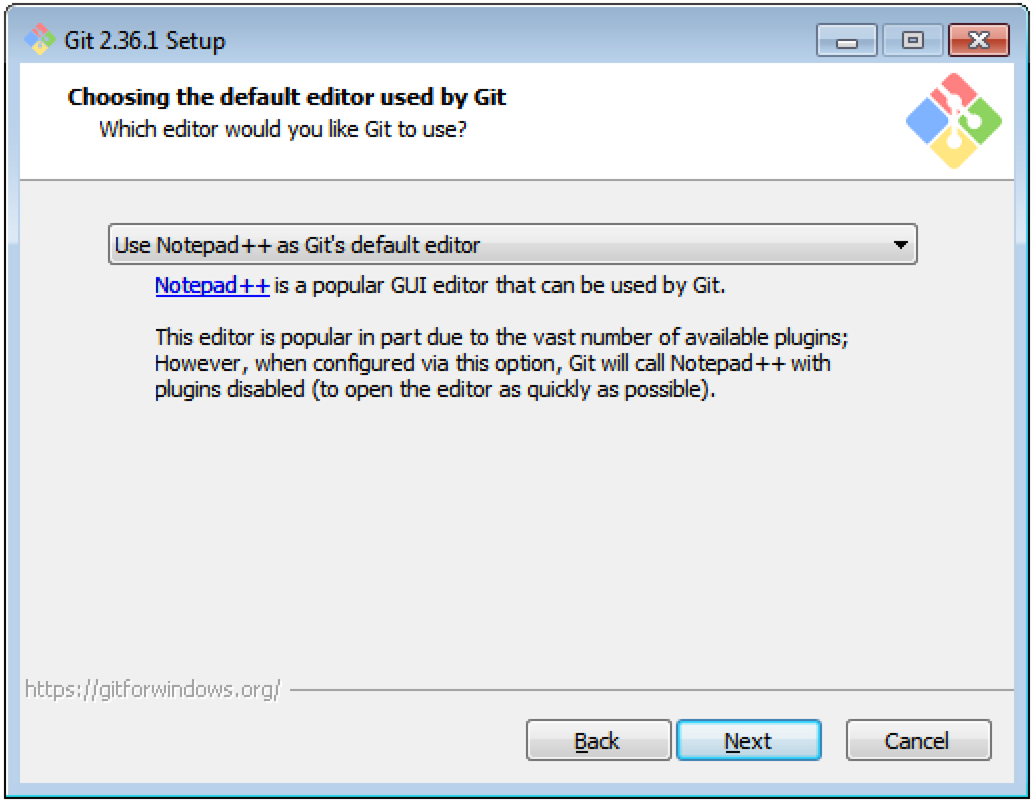
Click on Next Button



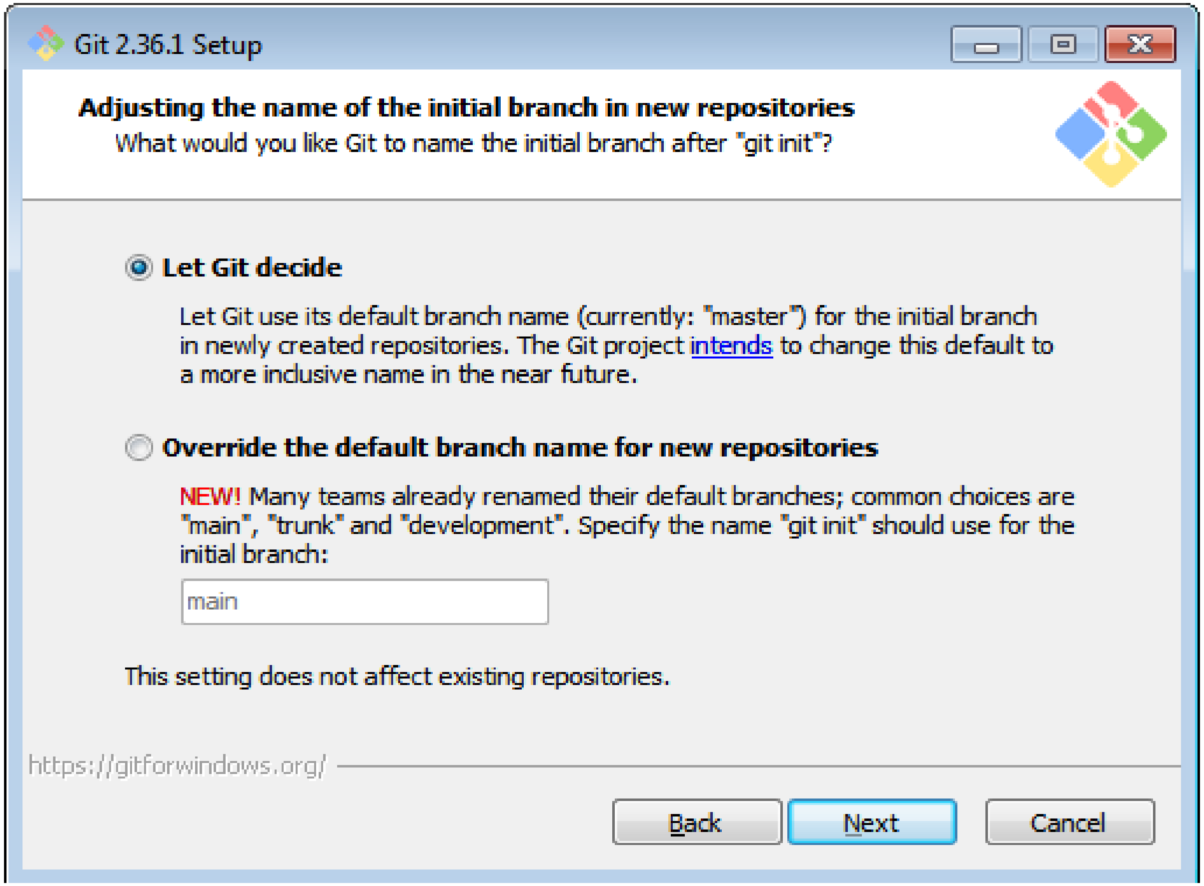
Click on Next Button



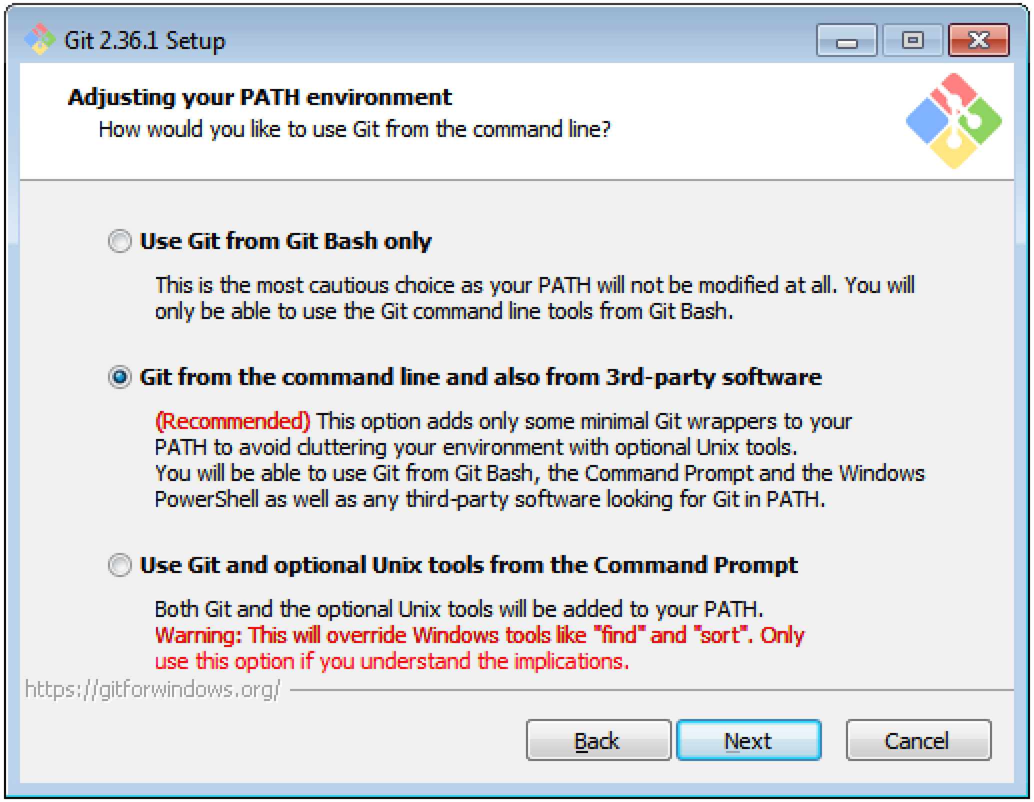
Click on Next Button



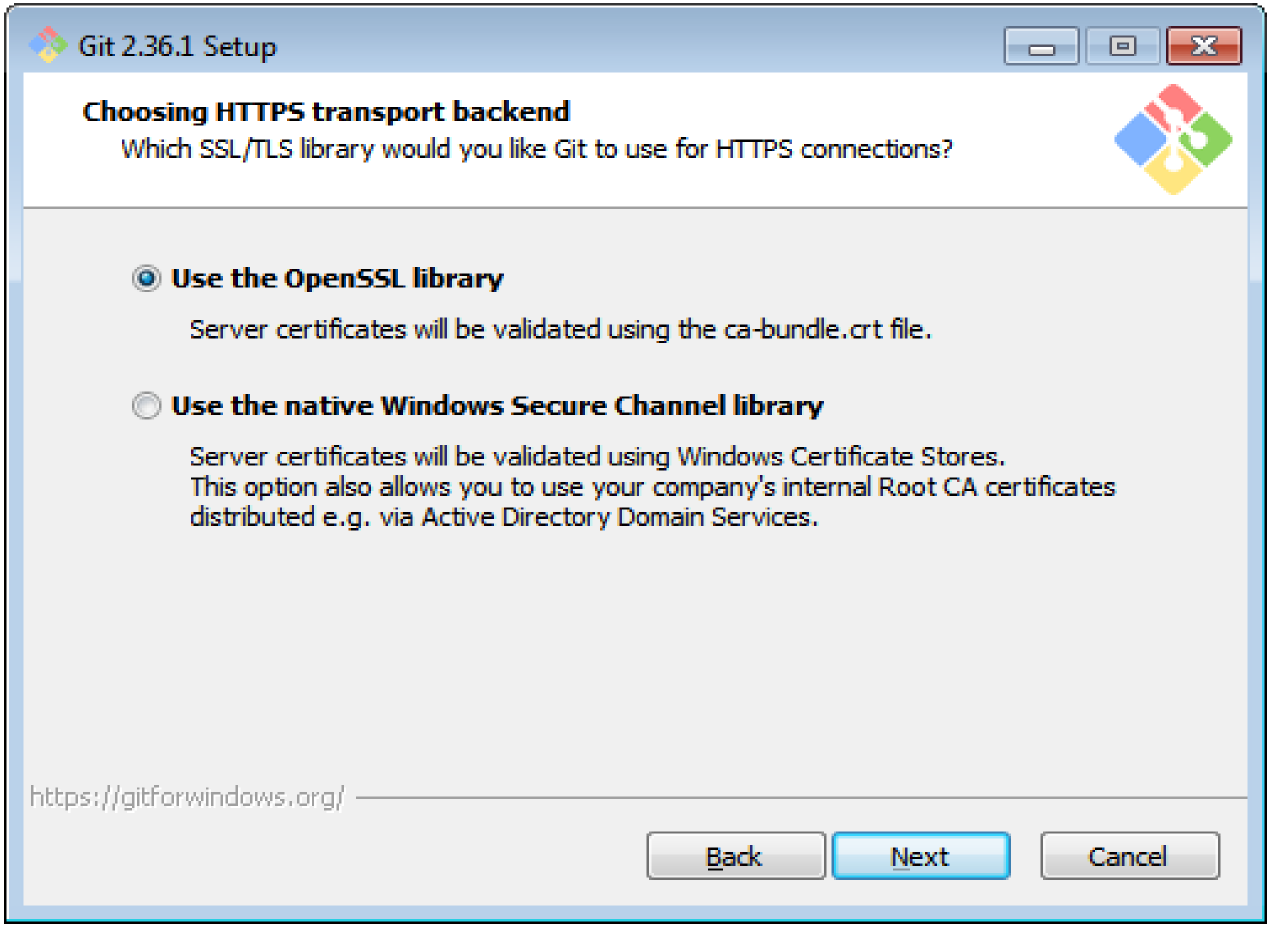
Select proper editor and click on Next button



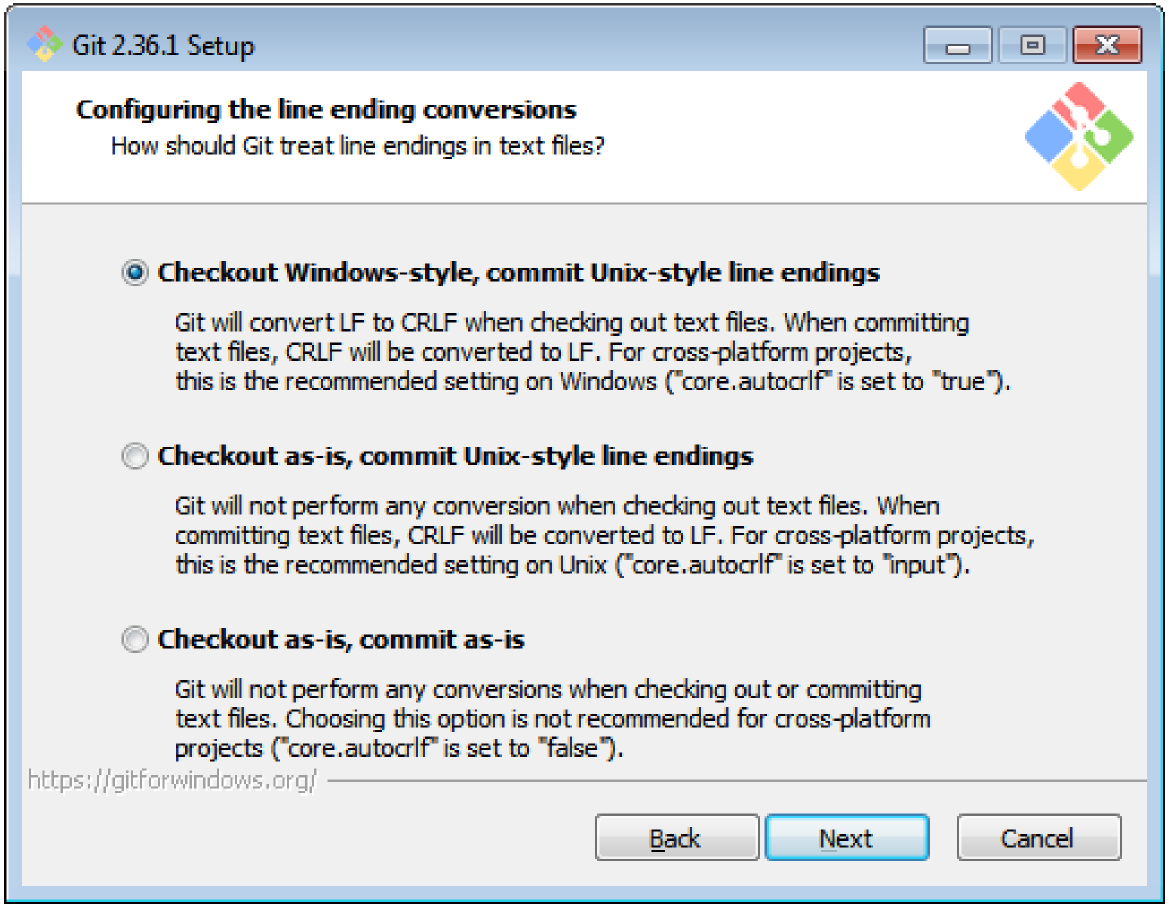
Select proper editor and click on Next button



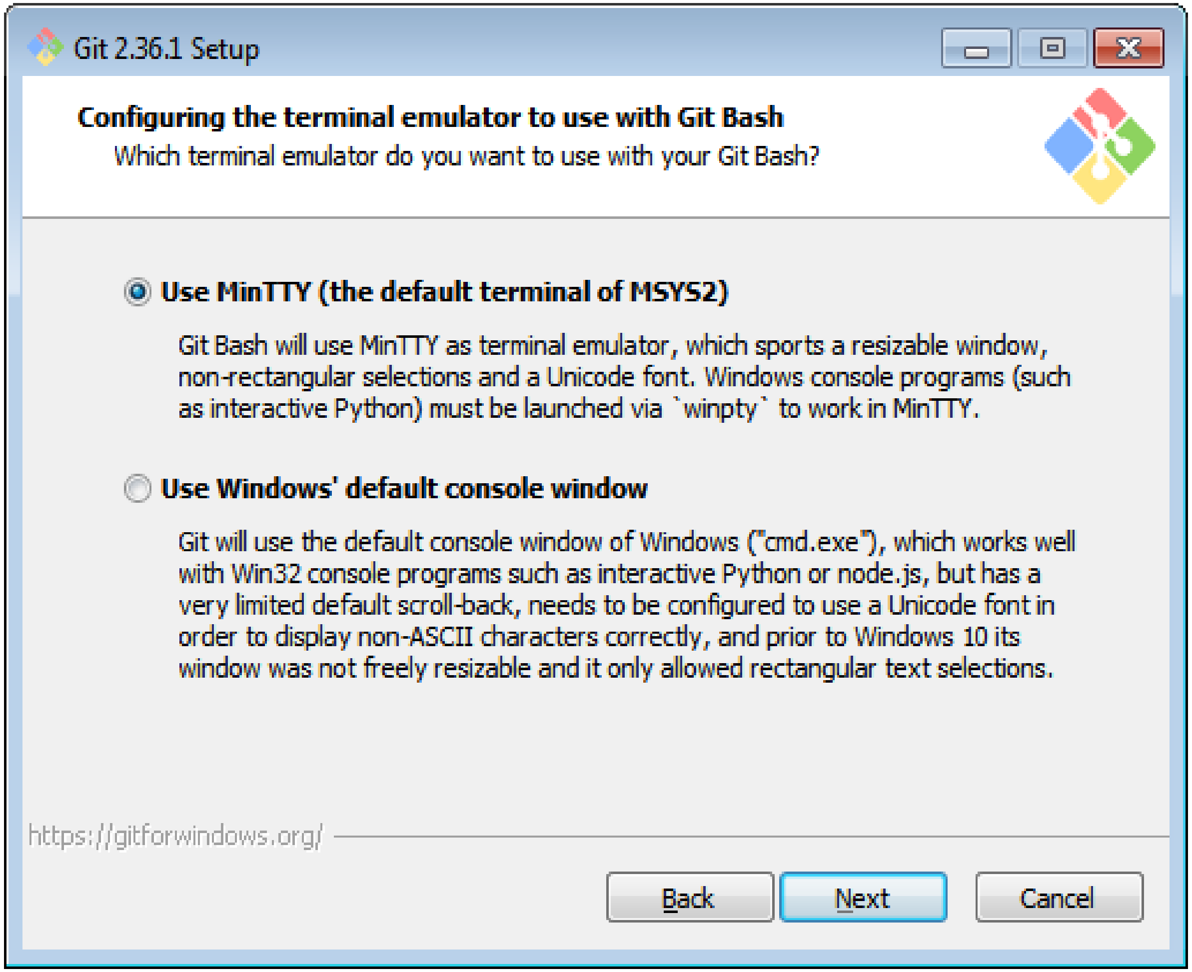
Click on Next button



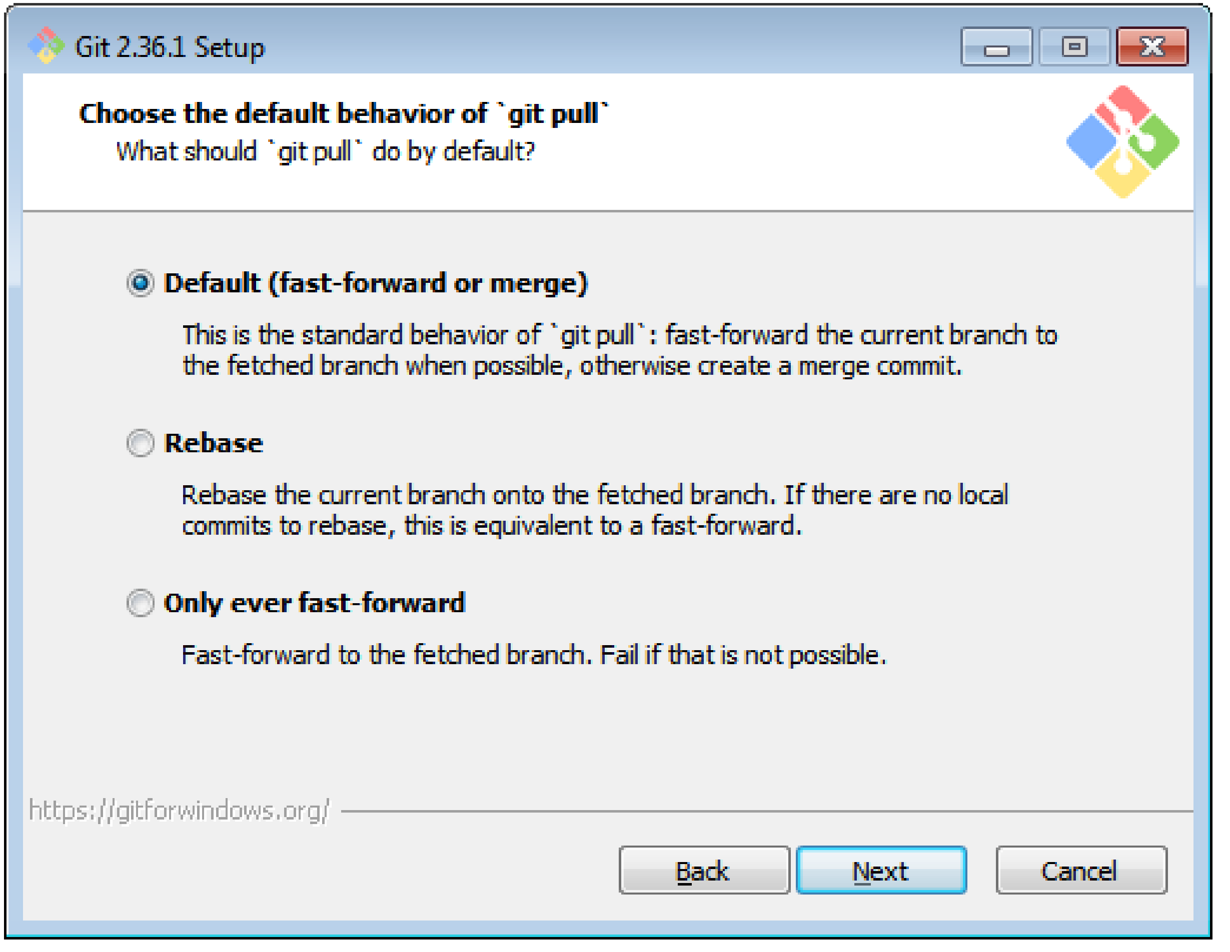
Click on Next button



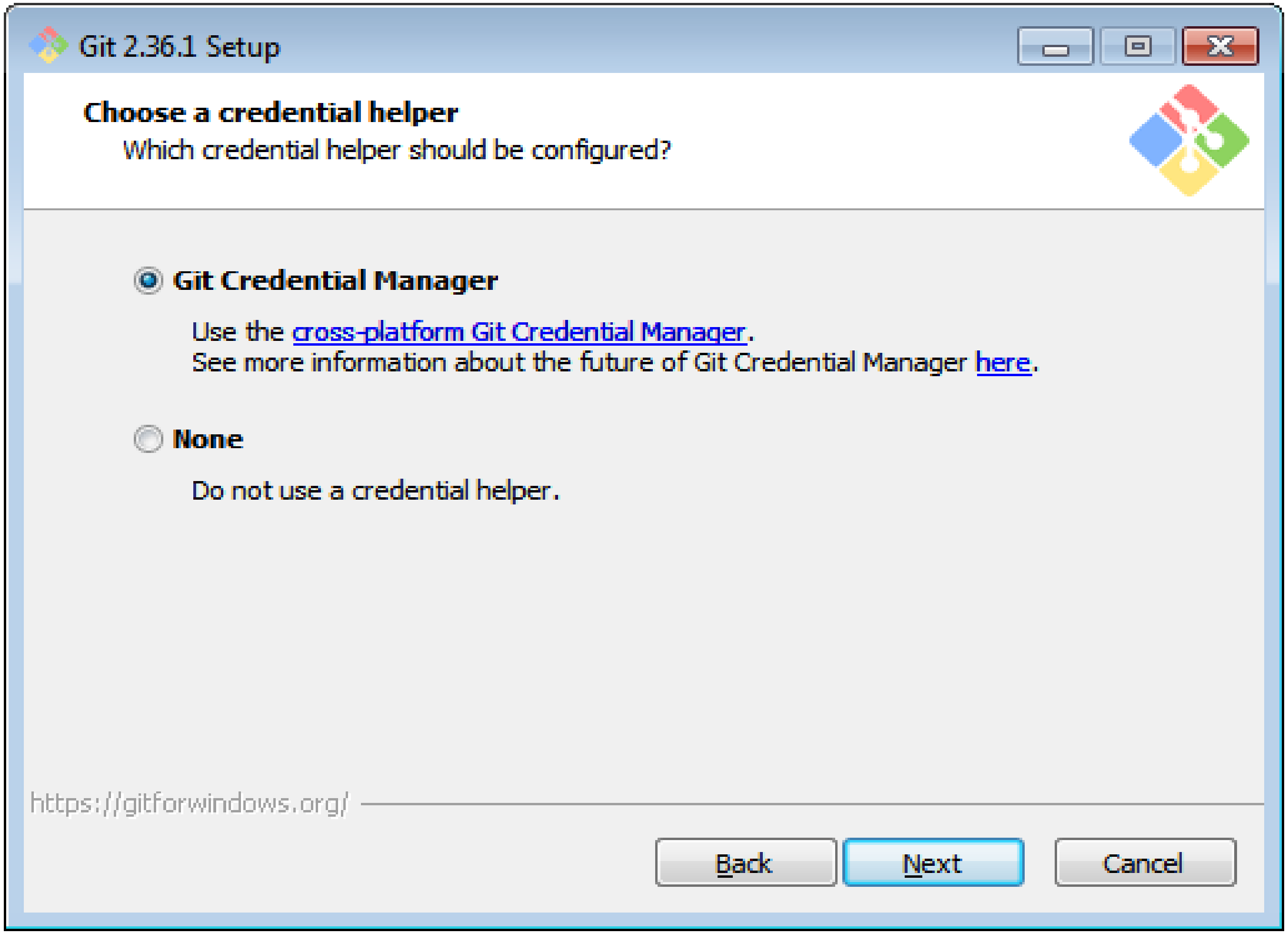
Click on Next button



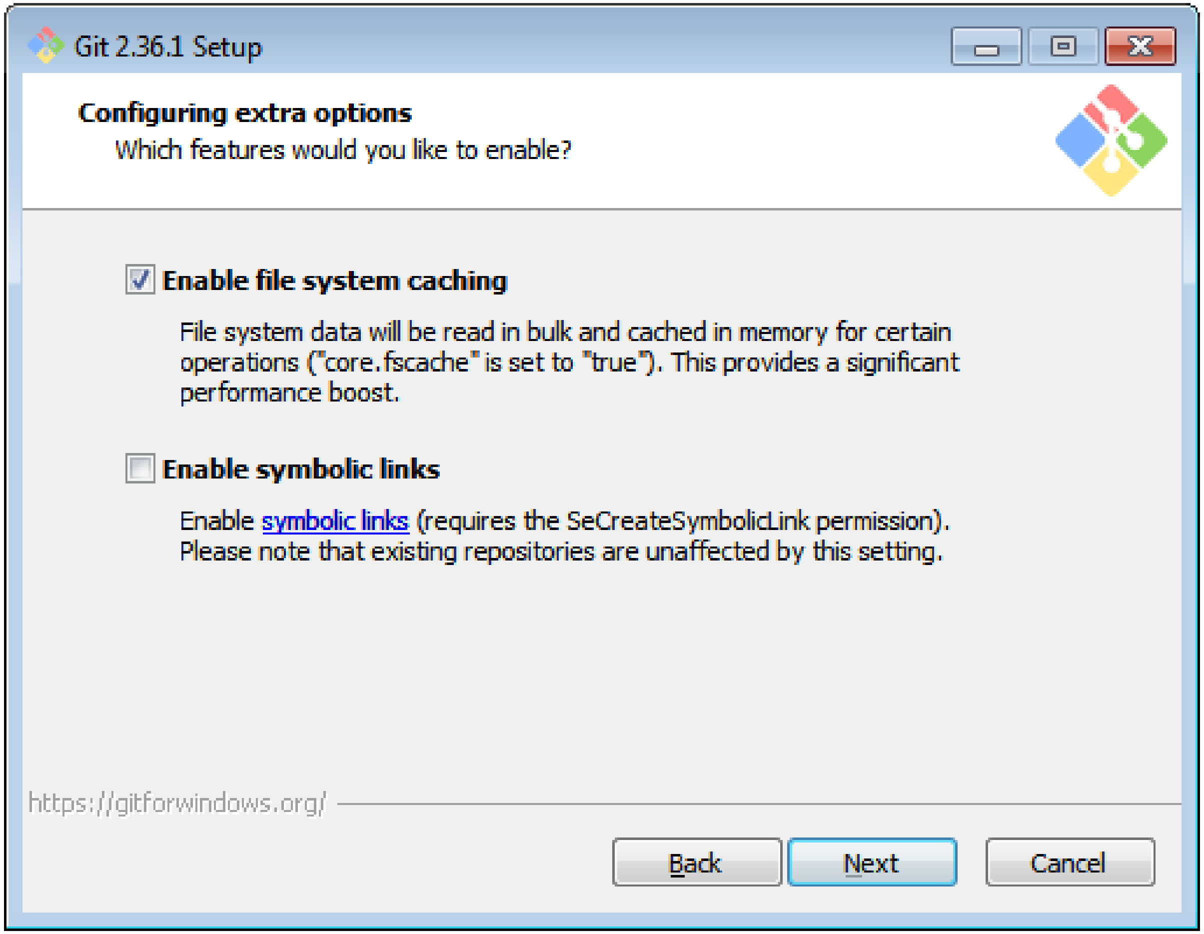
Click on Next button



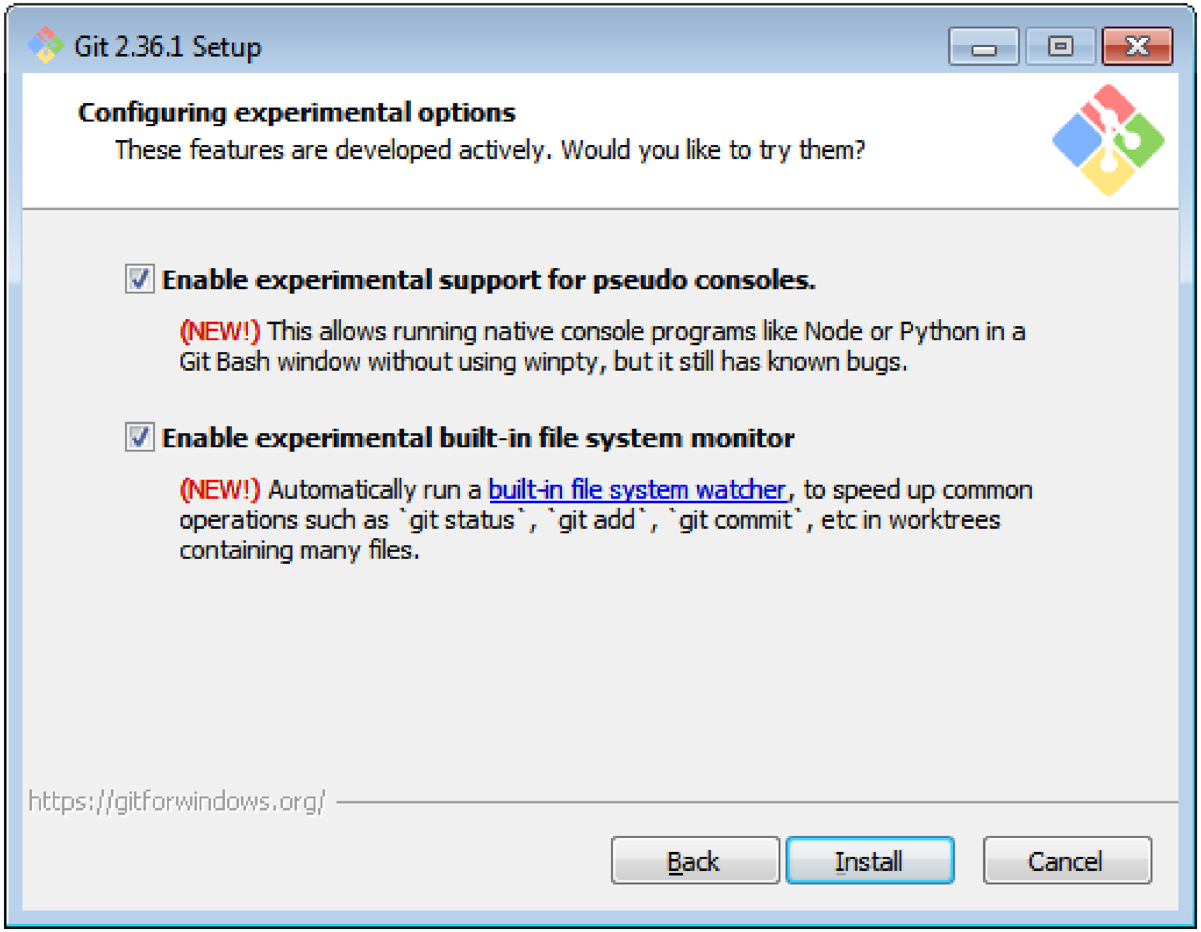
Click on Next button



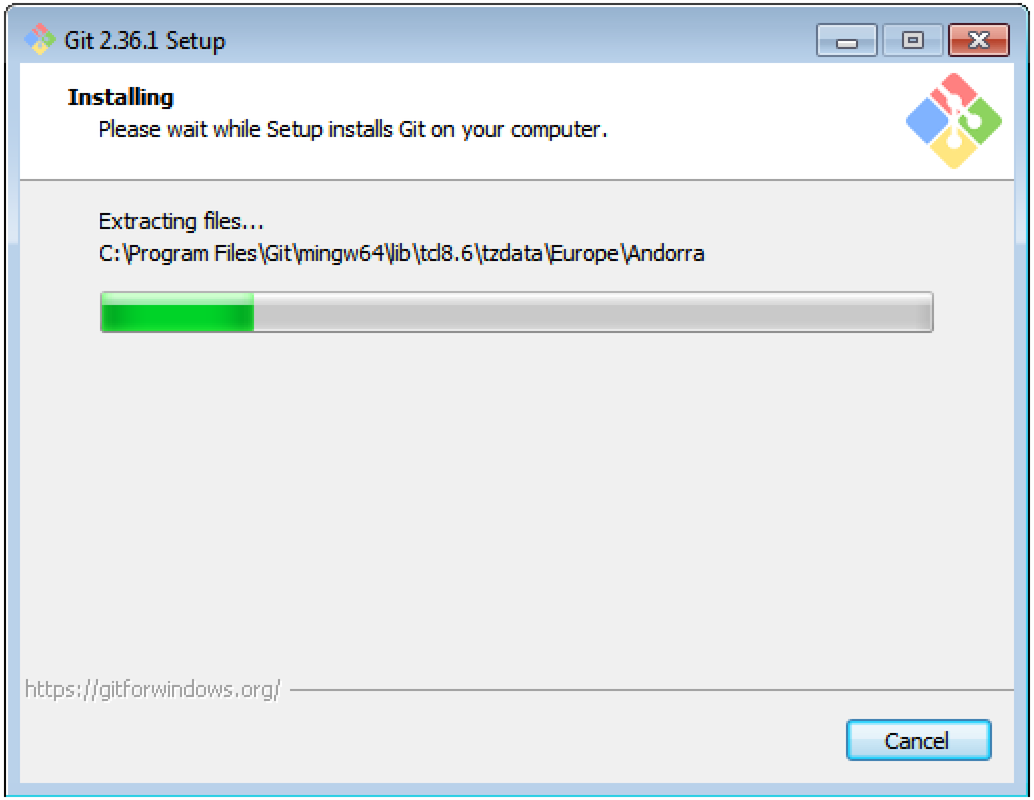
Click on Next button

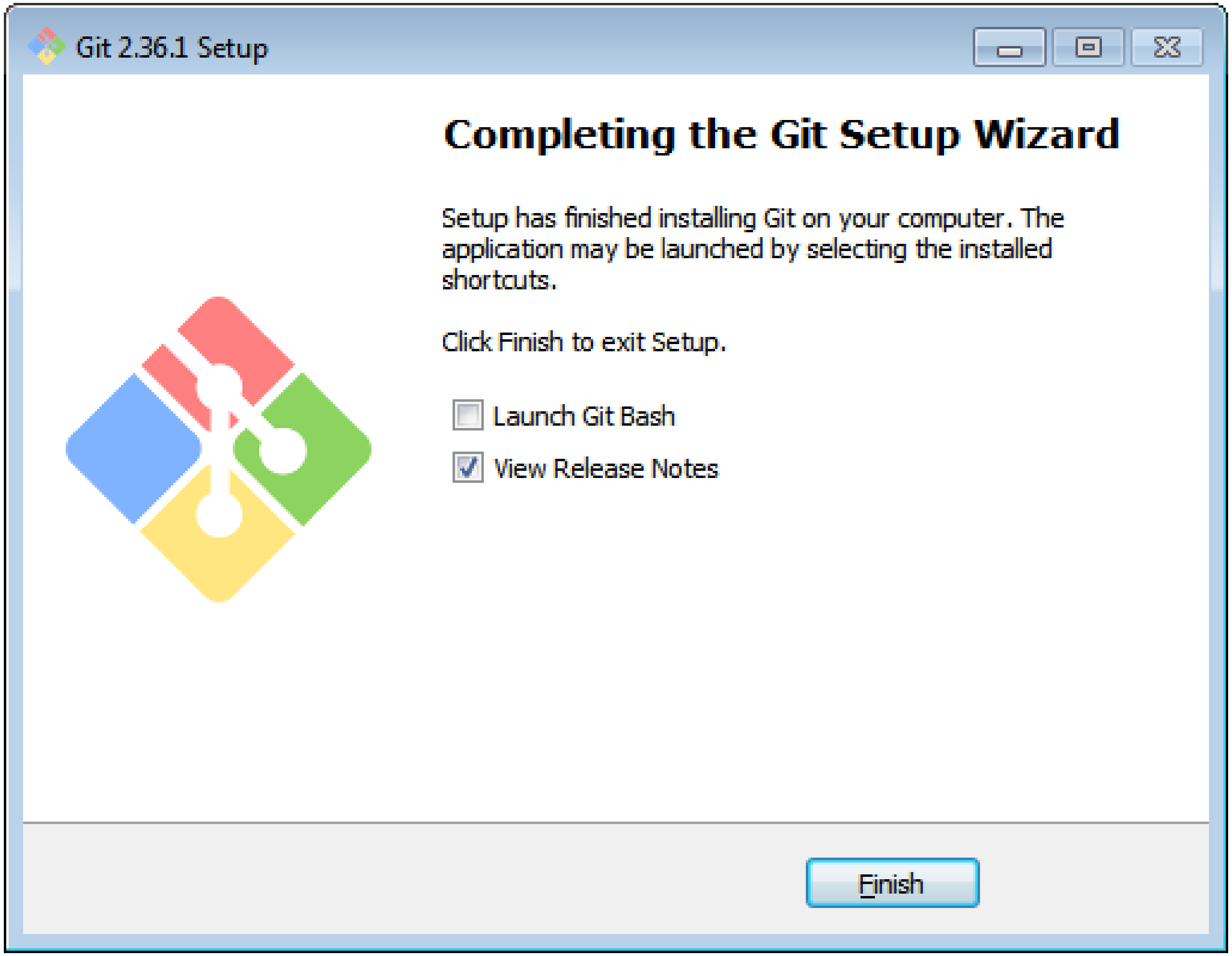


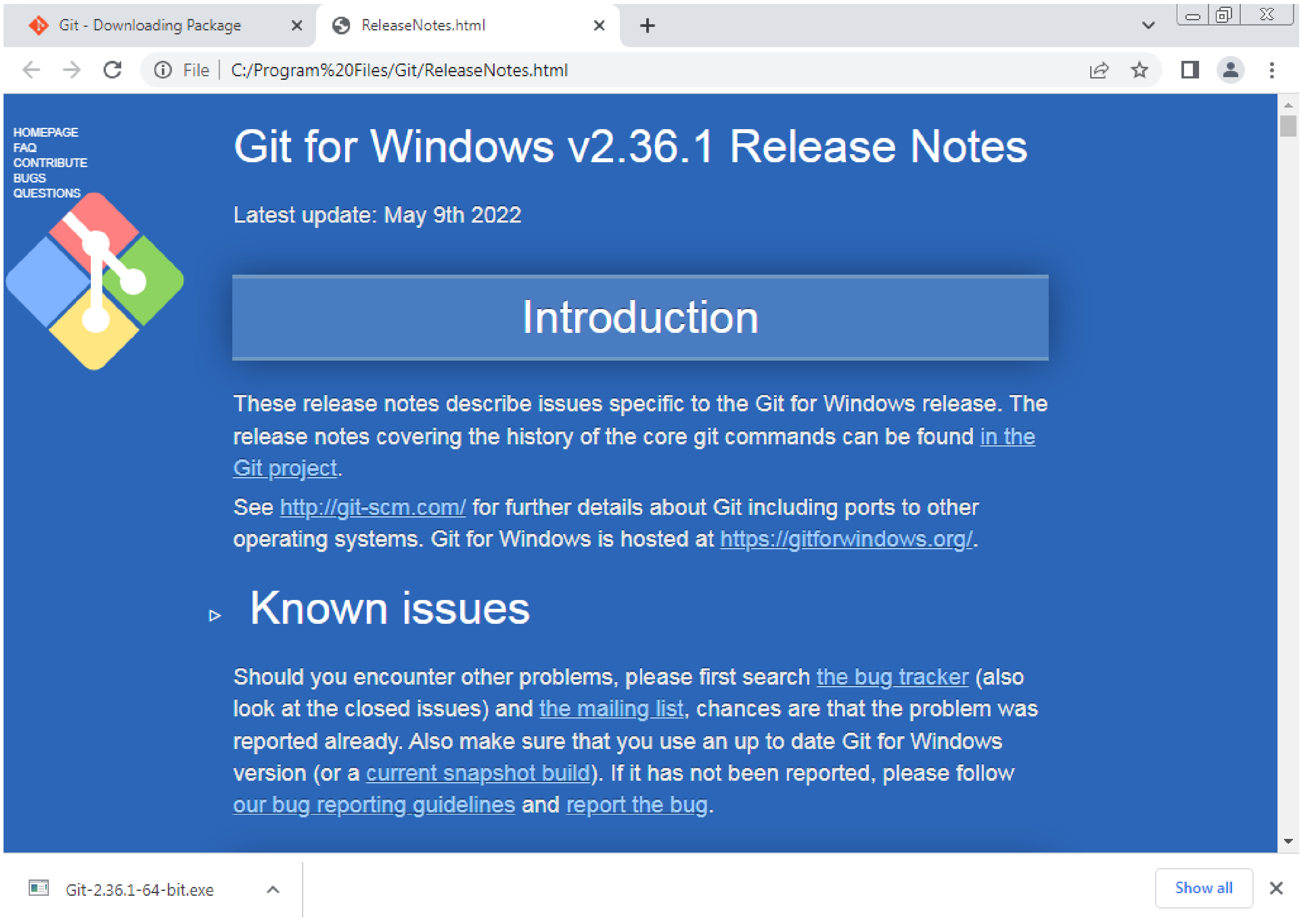
Click on Next button



Click on **Install** button


Git 2.36.1 Setup
Installing
Please wait while Setup installs Git on your computer.
Extracting files...
C:\Program Files\Git\mingw64\lib\tcl8.6\tzdata\Europe\Andorra
https://gitforwindows.org/
Cancel


Git 2.36.1 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
Finish



Git for Windows v2.36.1 Release Notes

Latest update: May 9th 2022

Introduction

These release notes describe issues specific to the Git for Windows release. The release notes covering the history of the core git commands can be found in the Git project.

See http://git-scm.com/ for further details about Git including ports to other operating systems. Git for Windows is hosted at https://gitforwindows.org/.

Known issues

Should you encounter other problems, please first search the bug tracker (also look at the closed issues) and the mailing list, chances are that the problem was reported already. Also make sure that you use an up to date Git for Windows version (or a current snapshot build). If it has not been reported, please follow our bug reporting guidelines and report the bug.

$ git config user.name

- Above command display user name

```
$ git config user.name

the director@computer169 MINGW64 ~
```

$ git config --list

- This command used to **list all the settings Git.**

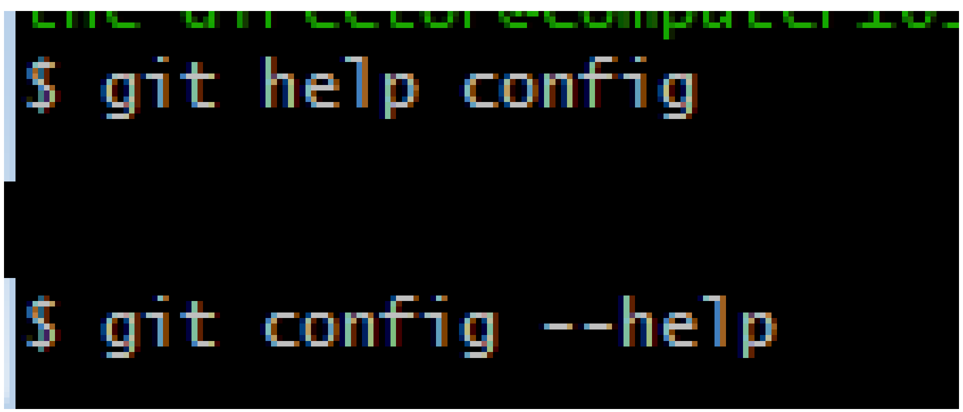
```
 git config --list
iff.astextplain.textconv=astextplain
ilter.lfs.clean=git-lfs clean -- %f
ilter.lfs.smudge=git-lfs smudge -- %f
ilter.lfs.process=git-lfs filter-process
ilter.lfs.required=true
ttp.sslbackend=openssl
ttp.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
ore.autocrlf=true
ore.fscache=true
ore.symlinks=false
ore.usebuiltinfsmonitor=true
ore.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -nosession -noPlugin
ull.rebase=false
redential.helper=manager-core
redential.https://dev.azure.com.usehttppath=true
nit.defaultbranch=master

he director@computer169 MINGW64 ~
```

Getting Help

$ git help <verb>

$ git <verb> --help



Git - Downloading Packag | ReleaseNotes.html | Installing Git + Initial Setu | git-config(1)

File | C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-config.html

git-config(1) Manual Page

NAME

git-config - Get and set repository or global options

SYNOPSIS

git config [<file-option>] [--type=<type>] [--fixed-value] [--show-origin] [--show-scope] [-z|--null] <name> [<value> [<value-pattern>]]
git config [<file-option>] [--type=<type>] --add <name> <value>
git config [<file-option>] [--type=<type>] [--fixed-value] --replace-all <name> <value> [<value-pattern>]
git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] --get <name> [<value-pattern>]
git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] --get-all <name> [<value-pattern>]
git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] [--name-only] --get-regexp <name-regex> [<value-pattern>]

Git-2.36.1-64-bit.exe

Show all

A quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the –h option, as in:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run               dry run
    -v, --verbose               be verbose

    -i, --interactive           interactive picking
    -p, --patch                 select hunks interactively
    -e, --edit                  edit current diff and apply
    -f, --force                 allow adding otherwise ignored files
    -u, --update                update tracked files
    --renormalize               renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add         record only the fact that the path will be added later
    -A, --all                   add changes from all tracked and untracked files
    --ignore-removal            ignore paths removed in the working tree (same as --no
-all)
    --refresh                   don't add, only refresh the index
    --ignore-errors             just skip files which cannot be added because of
errors
    --ignore-missing            check if - even missing - files are ignored in dry run
    --chmod (+|-)x              override the executable bit of the listed files
    --pathspec-from-file <file> read pathspec from file
    --pathspec-file-nul         with --pathspec-from-file, pathspec elements are
separated with NUL character
```

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose

    -i, --interactive     interactive picking
    -p, --patch           select hunks interactively
    -e, --edit            edit current diff and apply
    -f, --force           allow adding otherwise ignored files
    -u, --update          update tracked files
    --renormalize         renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add   record only the fact that the path will be added later
    -A, --all             add changes from all tracked and untracked files
    --ignore-removal      ignore paths removed in the working tree (same as --no
-all)
    --refresh             don't add, only refresh the index
    --ignore-errors       just skip files which cannot be added because of error
s
    --ignore-missing      check if - even missing - files are ignored in dry run
    --sparse              allow updating entries outside of the sparse-checkout
cone
    --chmod (+|-)x        override the executable bit of the listed files
    --pathspec-from-file <file>
                          read pathspec from file
    --pathspec-file-nul   with --pathspec-from-file, pathspec elements are separ
ated with NUL character
```

```
$ git config -h
usage: git config [<options>]

Config file location
    --global              use global config file
    --system              use system config file
    --local               use repository config file
    --worktree            use per-worktree config file
    -f, --file <file>     use given config file
    --blob <blob-id>      read config from given blob object

Action
    --get                 get value: name [value-pattern]
    --get-all             get all values: key [value-pattern]
    --get-regexp          get values for regexp: name-regex [value-pattern]
    --get-urlmatch        get value specific for the URL: section[.var] URL
    --replace-all         replace all matching variables: name value [value-pattern]
    --add                 add a new variable: name value
    --unset               remove a variable: name [value-pattern]
    --unset-all           remove all matches: name [value-pattern]
    --rename-section      rename section: old-name new-name
    --remove-section      remove a section: name
    -l, --list            list all
    --fixed-value         use string equality when comparing values to 'value-pattern'
    -e, --edit            open an editor
    --get-color           find the color configured: slot [default]
    --get-colorbool       find the color setting: slot [stdout-is-tty]

Type
    -t, --type <type>     value is given this type
    --bool                value is "true" or "false"
    --int                 value is decimal number
    --bool-or-int         value is --bool or --int
    --bool-or-str         value is --bool or string
    --path                value is a path (file or directory name)
    --expiry-date         value is an expiry date

Other
    -z, --null            terminate values with NUL byte
    --name-only           show variable names only
```

$git --version

- This command used to **display installed version of GIT**

```
$ git --version
git version 2.36.1.windows.1
```

Your Identity :

$ git config - -global user.name “Sangita Phunde”

$ git config - -global user.email sangita.phunde@rediffmail.com

```
$ git config --global user.name "Sangita Phunde"
```

```
$ git config --global user.email sangita.phunde@rediffmail.com
```

Your Identity :

Notice the "global" argument; it means that the setup is for all future Git repositories, so you don't have to set this up again in the future.

$ git config –global core.editior="notepad"

Your Identity :

You can find the file recording your Git configuration on your home folder.

Windows : c:\Users\YourName\.gitconfig

Your Identity :

$ git config - -list

```
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.usebuiltinfsmonitor=true
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -
nosession -noPlugin
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Sangita Phunde
user.email=sangita.phunde@rediffmail.com
```

Your Identity :

$ git config - -list

```
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.usebuiltinfsmonitor=true
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -
nosession -noPlugin
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Sangita Phunde
user.email=sangita.phunde@rediffmail.com
```

Your default branch name

By default Git will create a branch called ***master*** when you create a new repository with git init. From Git version 2.28 onwards, you can set a different name for the initial branch.

Your Identity :

By default branch name is Master

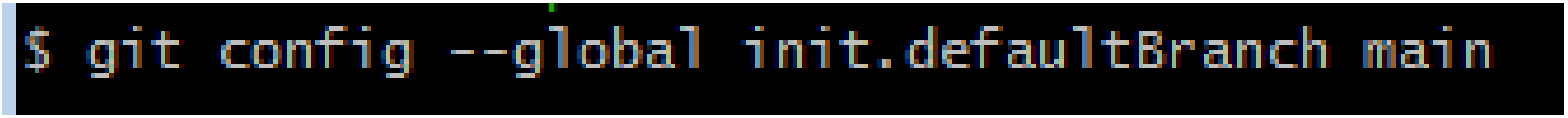
```
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.usebuiltinfsmonitor=true
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -
nosession -noPlugin
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Sangita Phunde
user.email=sangita.phunde@rediffmail.com
```

$ git config - -list

Your default branch name

- To set *main* as the default branch name do:

$ git config --global init.defaultBranch main



Your default branch name

```
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.usebuiltinfsmonitor=true
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -
nosession -noPlugin
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Sangita Phunde
user.email=sangita.phunde@rediffmail.com
init.defaultbranch=main
```

You can also check what Git thinks a specific key's value is by typing

git config <key>

$ git config user.name

$ git config user.email

$ git config init.defaultBranch

```
$ git config user.name
Sangita Phunde

the director@computer169 MINGW64 ~
$ git config user.email
sangita.phunde@rediffmail.com

the director@computer169 MINGW64 ~
$ git config init.defaultbranch
main

the director@computer169 MINGW64 ~
$ |
```

```
git config --global user.name "Mosh Hamedani"

git config --global user.email programmingwithmosh@gmail.com
```

Auto - Complete

When you start typing a command or an argument to a command, Git has a helpful auto completion feature for two things :

- Provide valid values
- Automatically complete the commands

Auto - Complete

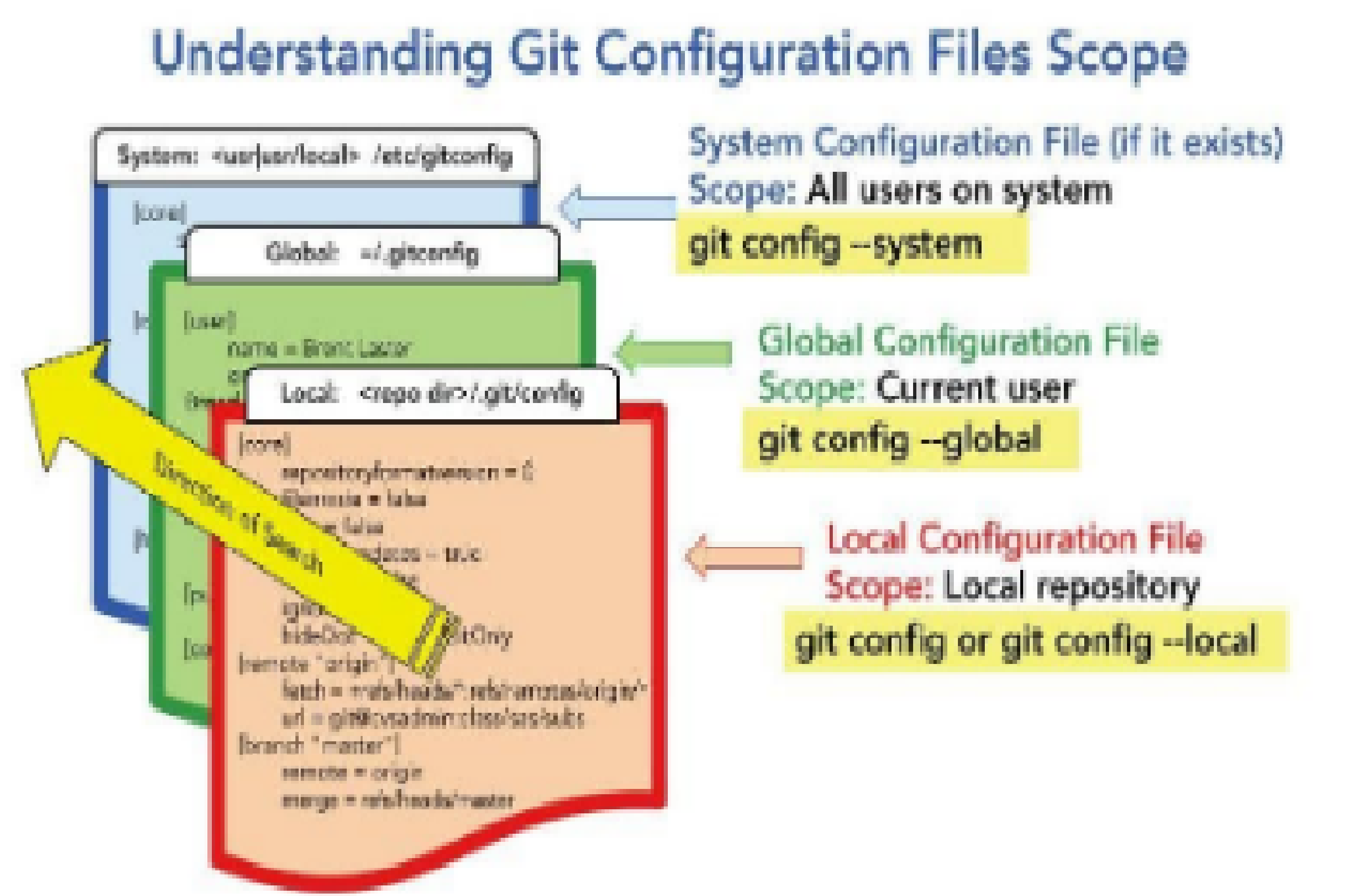
$ git c <tab> <tab>

```
$ git c
checkout                              clone
cherry                                commit
cherry-pick                           config
citool                                credential-helper-selector
clean                                 credential-manager-core
```

Auto - Complete

$ git co <tab> <tab>

```
the director@computer169 MINGW64 ~
$ git co
commit    config
```


Understanding Git Configuration Files Scope
System: <usr|usr/local> /etc/gitconfig
[core]
Global: ~/.gitconfig
[user]
Local: <repo dir>/.git/config
[core]
[remote "origin"]
[branch "master"]
remote = origin
merge = refs/heads/master
Direction of Search
System Configuration File (if it exists)
Scope: All users on system
git config --system
Global Configuration File
Scope: Current user
git config --global
Local Configuration File
Scope: Local repository
git config or git config --local

Getting a Git Repository

Repositories :

A repository is a storage, where all your project and all the changes made to it are kept.

It is only normal folder on your system, so it is very easy to manipulate.

Repositories :

So for each project you want to start, you should

-> Create the directory containing your project

-> Navigate into the directory

-> Initialize a Git repository

Getting a Git Repository :

You typically obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or

2. You can *clone* an existing Git repository from elsewhere.

Initializing a Repository in an Existing Directory :

If you have a project directory that is currently **not under version control and you want to start controlling it with Git, you first need to go to that project's directory.**

you've never done this, it looks a little different depending on which system you're running:

Git Init :

- The **git init** command is the first command that you will run on Git.
- The git init command is used **to create a new blank repository.**
- It is used to make an existing project as a Git project.
- Several Git commands run inside the repository, **but init command can be run outside of the repository.**

Git Init :

- The git init command creates a **.git** subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata.

- These metadata can be categorized into objects, refs, and temp files.

- It also initializes a HEAD pointer for the master branch of the repository.

Initializing a Repository in an Existing Directory :

for Windows:

```
$ cd C:/Users/user/my_project
```

and type:

```
$ git init
```

Initializing a Repository in an Existing Directory :

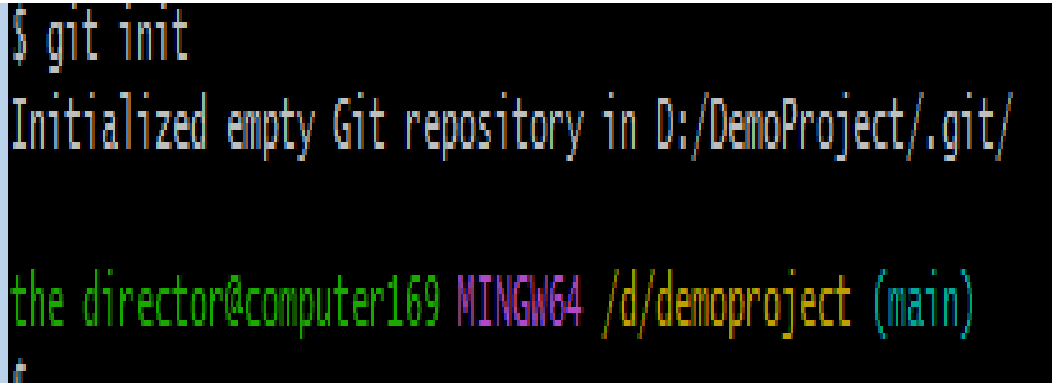
for Windows:

$ cd C:/Users/user/my_project

Initializing a Repository in an Existing Directory :

for Windows:

$ git init



This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton.

```
$ mkdir Myproject
```

```
$ cd Myproject/
```

```
the director@computer169 MINGW64 ~/Myproject
```

```
$ mkdir mynewproject
$ cd mynewproject/
$ git init
```

Git will create a directory called ".git" that will contain all your changesets and snapshots.

You should also know that initializing is the only way to get repository. You can copy an entire repository with all its history and snapshots. It is called “cloning”.

Working Directory :

What about the empty area outside the “.git” directory? It is called the working directory, and the files you will be working on will be stored there.

Working Directory :

Generally, your most recent version will be on the **Working Directory**.

Working Directory :

Each file you work on is on the Working Directory. There is nothing particular about this place except the fact that you will only manipulate the files here directly.

Never modify the files inside the “.git” directory

Working Directory :

Each file you work on is on the Working Directory. There is nothing particular about this place except the fact that you will only manipulate the files here directly.

Never modify the files inside the “.git” directory

Working Directory :

Git will detect any new file you will place in the **Working Directory**.

Working Directory :

You can check the status of the directory by using

$git status

Staging Area

Staging Area :

The Staging area is where your files go before the snapshots are taken. Not every file you modified on the Working Directory should be taken into account when taking a snapshot of the current state of the project.

Only the files placed in the Staging Area will be snapshotted.

Staging Area :

So, before taking a snapshot of the project, you select which changed files to take account of. A change in a file can be creating, deleting, or editing.


SYSTEM
All users
GLOBAL
All repositories of the current user
LOCAL
The current repository

Recording Changes to the Repository

Remember that each file in your working directory can be in one of two states: ***tracked* or *untracked*.**

Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be **unmodified, modified, or staged**. In short, tracked files are files that Git knows about.

Recording Changes to the Repository

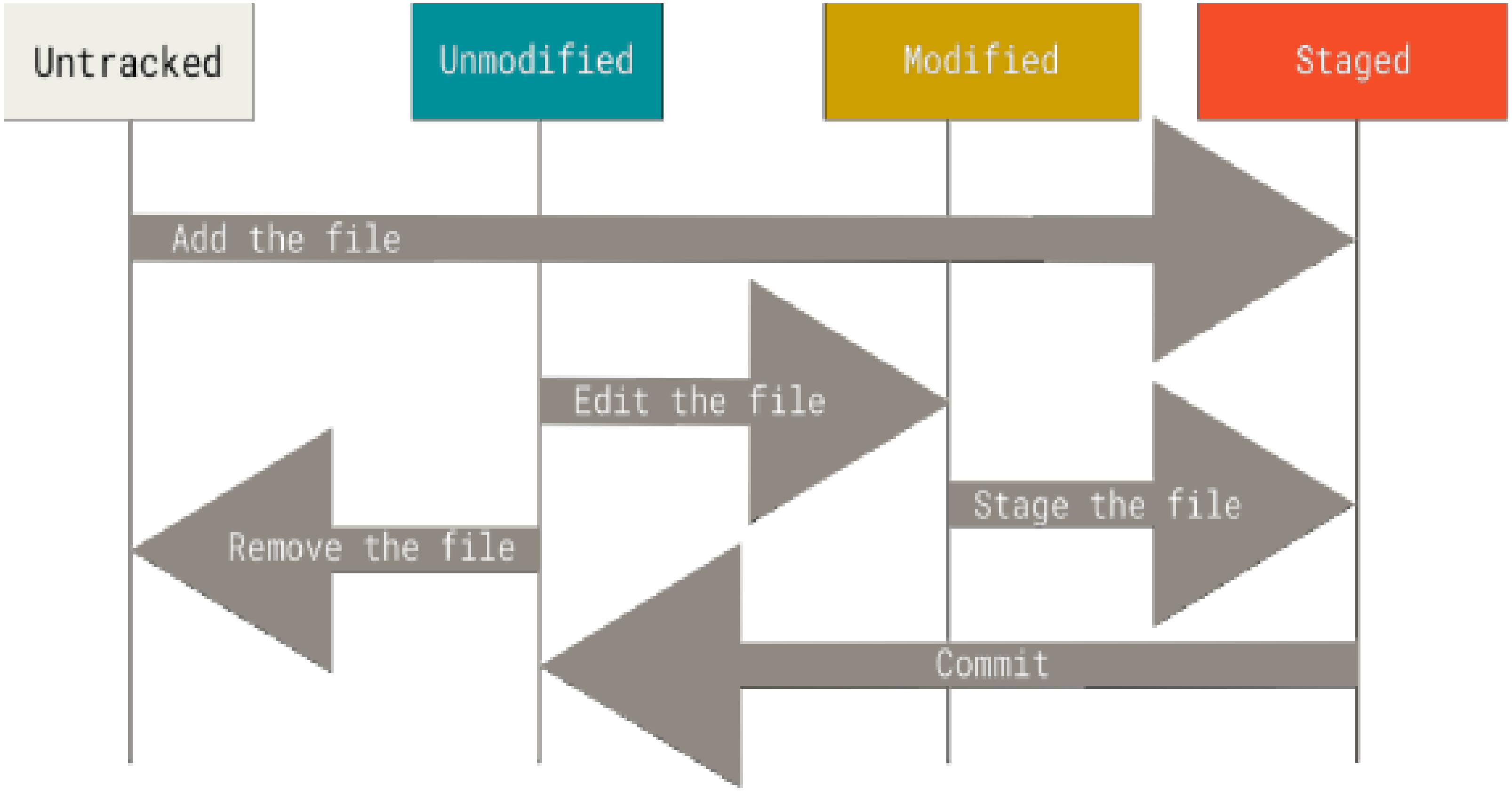
Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.

When you **first clone a repository**, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

Recording Changes to the Repository

Git sees them as modified, because you've changed them since your last commit.

As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit

The main tool you use to determine which files are in which state is the git status command.

$ git status

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean.
```

$ git status

On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean.

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn’t see any untracked files, or they would be listed here. Finally, the command tells you which branch you’re on and informs you that it has not diverged from the same branch on the server. For now, that branch is always **master**, which is the default.

$ cd d:

$ cd demoproject

$ git status

```
the director@computer169 MINGW64 ~
$ cd d:

the director@computer169 MINGW64 /d
$ cd demoproject

the director@computer169 MINGW64 /d/demoproject (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)

the director@computer169 MINGW64 /d/demoproject (main)
$
```

$ echo 'My Project' > README
$ git status

```
$ echo 'My project '>readme

the director@computer169 MINGW64 /d/demoproject (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main

No commits yet

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

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

You can see that your new **README file is untracked,** because it’s under the **“Untracked files”** heading in your status output.

Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit), and which hasn’t yet been **staged;**

Git won’t start including it in your commit snapshots until you explicitly tell it to do so.

It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let’s start tracking the file.

Tracking New Files

Tracking New Files :

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

$ git add README

To add a file to the Staging Area. Use the git command **“add”**.

$ git status

```
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main

No commits yet

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

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

the director@computer169 MINGW64 /d/demoproject (main)
$ 
```

$ git commit

```
$ git commit
```

$ git status

```
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main

No commits yet

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


the director@computer169 MINGW64 /d/demoproject (main)
$
```

The above command will prompt a default editor and ask for a commit message. We have made a change to **readme file** and want it to commit it. It can be done as follows:

As we run the command, it will prompt a default text editor and ask for a commit message. The text editor will look like as follows:

The above command will prompt a default editor and ask for a commit message. We have made a change to **readme file** and want it to commit it. It can be done as follows:

```
$ git commit
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
[main (root-commit) 5e055ec] hello  how are you
 1 file changed, 1 insertion(+)
 create mode 100644 readme
```

```
$ echo ' First file'>one

the director@computer169 MINGW64 /d/demoproject (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        one

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

the director@computer169 MINGW64 /d/demoproject (main)
$ git commit
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        one

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

the director@computer169 MINGW64 /d/demoproject (main)
$ git add one
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
warning: LF will be replaced by CRLF in one.
The file will have its original line endings in your working directory

the director@computer169 MINGW64 /d/demoproject (main)
$ git commit
```

```
$ git commit
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
[main aa55003] now i want to commit file here
 1 file changed, 1 insertion(+)
 create mode 100644 one
```

If you created multiple files, you can add them one after another or together like :

$ git add file1 file2 file3

Git commit –m :

The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor. It will run as follows:

$ git commit -m "Commit message."

```
$ echo ' First file'>second

the director@computer169 MINGW64 /d/demoproject (main)
$ git add second
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
warning: LF will be replaced by CRLF in second.
The file will have its original line endings in your working directory

the director@computer169 MINGW64 /d/demoproject (main)
$ git commit -m ' second file now update it'
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
[main fde058d]  second file now update it
 1 file changed, 1 insertion(+)
 create mode 100644 second

the director@computer169 MINGW64 /d/demoproject (main)
```

Git Commit Amend (Change commit message) :

The amend option lets us to edit the last commit. If accidentally, we have committed a wrong commit message, then this feature is a savage option for us. It will run as follows:

$ git commit --amend

Git Commit Amend (Change commit message) :

```
$ echo 'welcome'>wel.txt

$ git add wel.txt

$ git commit wel.txt

$ echo 'wel wel wel'>>wel.txt

$ git commit wel.txt --amend
```

```
$ git log
commit 698479f69eecaaadd81fabd0ecf72b360abc39b4 (HEAD -> main)
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:35:11 2022 +0530

    welcome for commit

    welcome amend
```

- **git add <file>**
- **git add <directory>**

Demo Folder

```
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hello.c
        new file:   one.c
        new file:   th.c
        new file:   th1.c
        new file:   two.c

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    hello.c

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

$ git add demo

```
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   demo/wel.txt
        modified:   hello.c
        new file:   one.c
        new file:   th.c
        new file:   th1.c
        new file:   two.c
```

$ git add demo

```
$ git commit demo -m 'folder commited'
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonit
hint: Disable this message with "git config advice.useCoreFSMonitorConfig
warning: LF will be replaced by CRLF in demo/wel.txt.
The file will have its original line endings in your working directory
[main d61e951] folder commited
 1 file changed, 2 insertions(+)
 create mode 100644 demo/wel.txt
```

EXAMPLES

EXAMPLE - 1

$ echo 'hello......'> hello.c

$ git rm goodbye.c

$ git add hello.c

$ git commit

The **git rm** command removes a file from a Git repository. This command removes a file from your file system and then removes it from the list of files tracked by a Git repository.

EXAMPLE - 1

$ git rm –cached READE.md

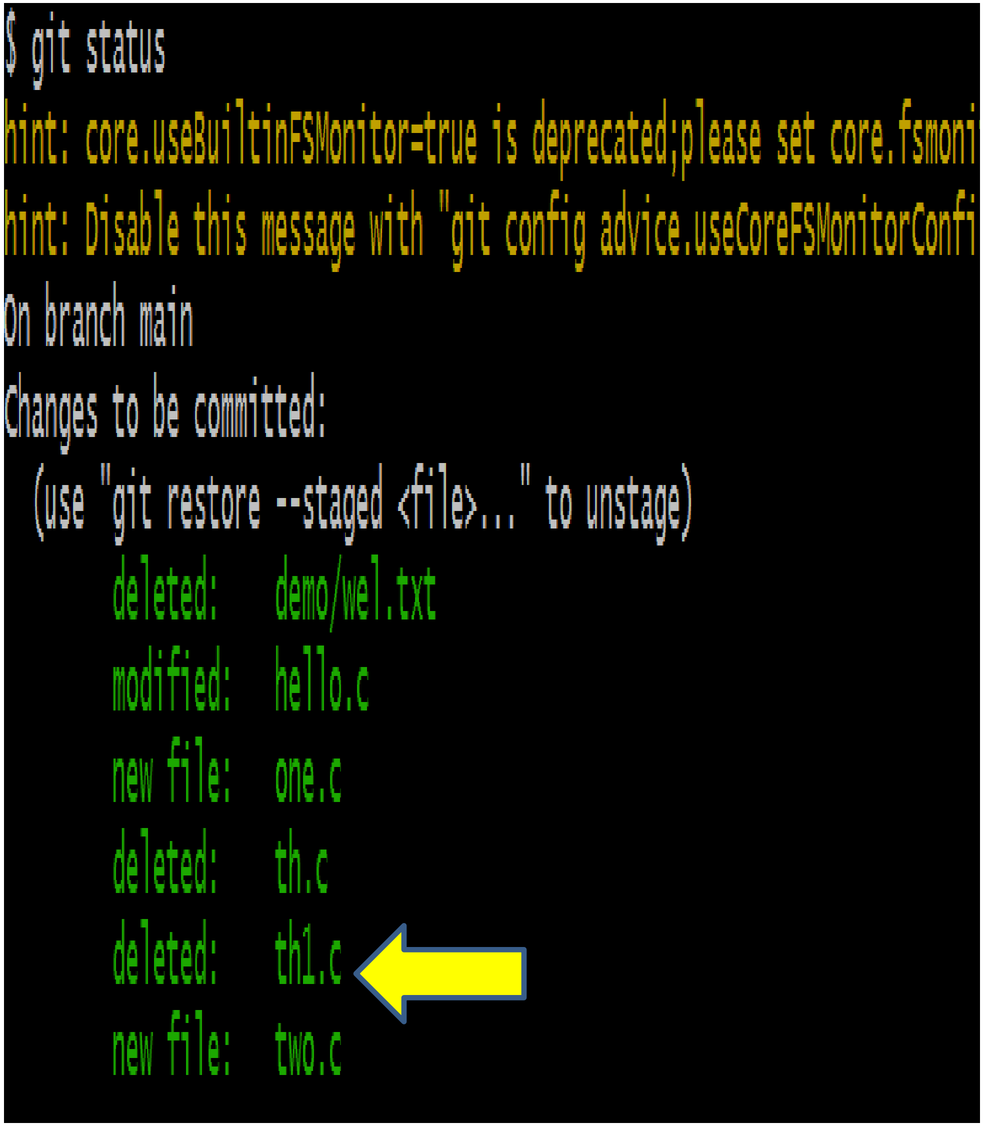
The **--cached** flag lets you delete a file from a Git repository without deleting it on your file system.

Caution : Don’t forget the option “—cached” when unstaging a file. If you forget it, you could lose your file !

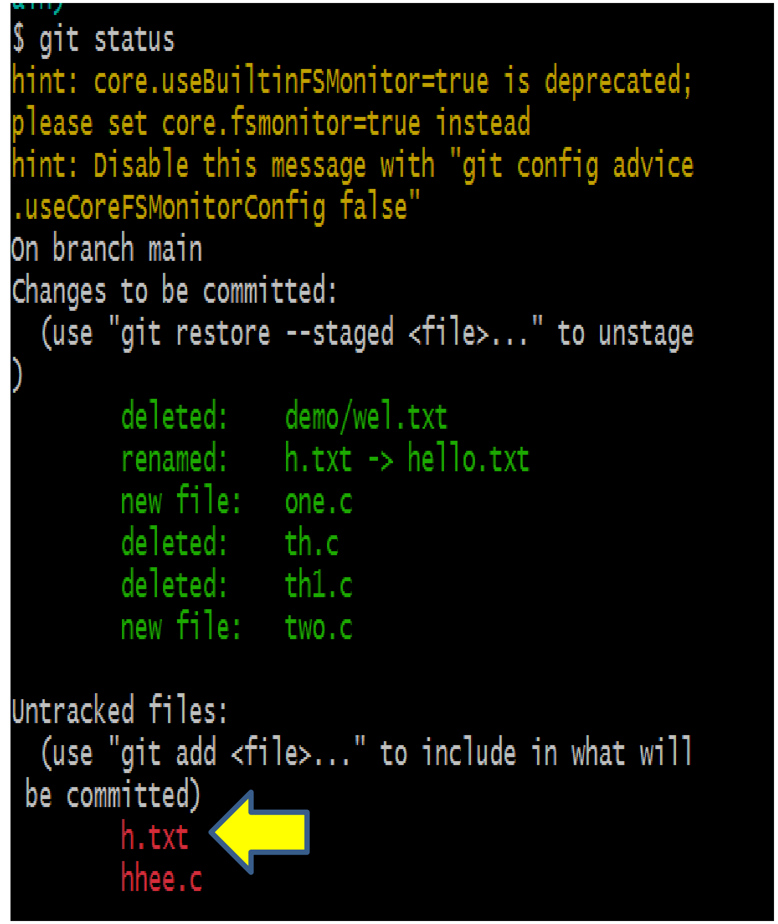
EXAMPLE - 1

$ git rm th1.c

File removed from actual location also.



EXAMPLE - 2



```
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;
please set core.fsmonitor=true instead
hint: Disable this message with "git config advice
.useCoreFSMonitorConfig false"
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage
)
        deleted:    demo/wel.txt
        renamed:    h.txt -> hello.txt
        new file:   one.c
        deleted:    th.c
        deleted:    th1.c
        new file:   two.c

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

$ echo 'hello......'> h.txt

$ git status

$ git add h.txt

$ git commit

$ git status

$ git rm --cached h.txt

$ git status

After commit file remove and send it back to working area.

By using –cached option.

EXAMPLE - 4

$ git add one.txt two.txt

$ git commit one.txt

After staging changes to many files, you can alter the order the changes are recorded in, by giving pathnames to git commit. When pathnames are given, the command makes a commit that only records the changes made to the named paths:

The summary of the commit will contain a lot of information :

- **The current branch : master**
- **The name of the previous commit : root-commit because this is our first commit**
- **The name of the commit : The first seven letters of the commit hash**
- **The commit message**
- **The number of files changed : one file**
- **The operation done to each file : creation.**

Tutorial

- What is Distributed Version Control System ? [5]

- How do you configure a Git repository to run code sanity checking tools right before making commits and preventing them if test fails [10]

Tutorial

- Explain the use of following GIT commands

  git add, git branch, git pull, git push, git clean [5]

- What is the difference between git pull and git fetch [5]

git init


This command turns a directory into an empty Git repository
Directory
Git Repository

Git Basic Commands

git add

Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area)

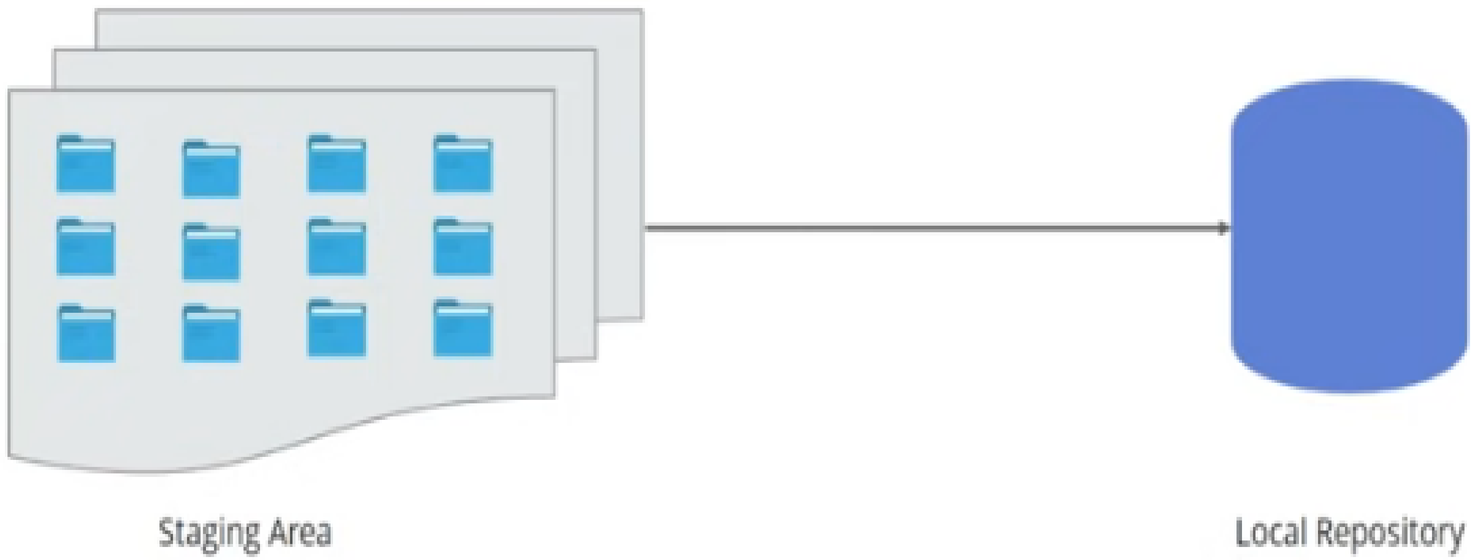
$ git add .

- Add all untracked files to staging area.

Git Basic Commands

git commit

Records the changes made to the files in a local repository

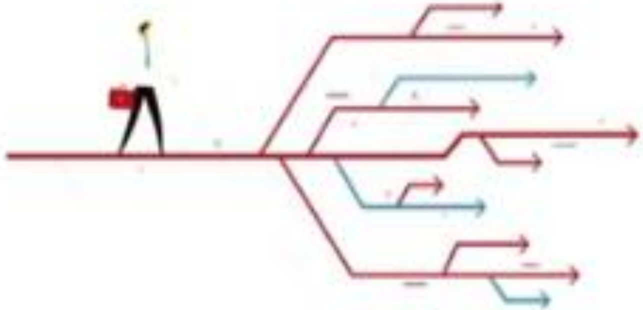


For easy reference, each commit has a unique ID

Git Basic Commands

git status

This command returns the current state of the repository



Returns current working branch

If a file is in the staging area, but not committed, it shows with *git status*

If there are no changes it'll return *nothing to commit, working directory clean.*

Git Basic Commands

git config

With Git, there are many configurations and settings possible. *git config* is how to assign these settings. Two important settings are user user.name and user.email

$git config user.name "sangita Priya"

```
$ git config user.name "priya sangita"

the director@computer169 MINGW64 /d/demoproject (main)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.usebuiltinfsmonitor=true
core.editor="C:\\Program Files\\Notepad++\\notepad++.exe" -multiInst -notabbar -nosession -
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Sangita Phunde
user.email=sangita.phunde@rediffmail.com
init.defaultbranch=main
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
user.name=priya sangita
```

Viewing the Commit History :

- **Git log** is a utility tool to review and read a history of everything that happens to a repository. Multiple options can be used with a git log to make history more specific.

- **Generally, the git log is a record of commits.**

Viewing the Commit History :

$ git log

press the **q (Q for quit)**. It will quit you from the situation and back you to the command line.

```
$ git log
commit fa85e0e69f48e93175671e0b98548d5ad416b429 (HEAD -> main)
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:11:35 2022 +0530

    ggggggg

commit 20647085cfd2a5c63e45276b8e16ed5b4cc282ac
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:08:02 2022 +0530

    sssssss

commit 6b07d2ac3d784eb16475829f5258f20debe6a26a
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:57:31 2022 +0530

    zfdfddsf

commit aa52d946b35bfd12558362425d6448ad95522d88
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:56:27 2022 +0530

    sssss
:...skipping...
commit fa85e0e69f48e93175671e0b98548d5ad416b429 (HEAD -> main)
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:11:35 2022 +0530

    ggggggg

commit 20647085cfd2a5c63e45276b8e16ed5b4cc282ac
```

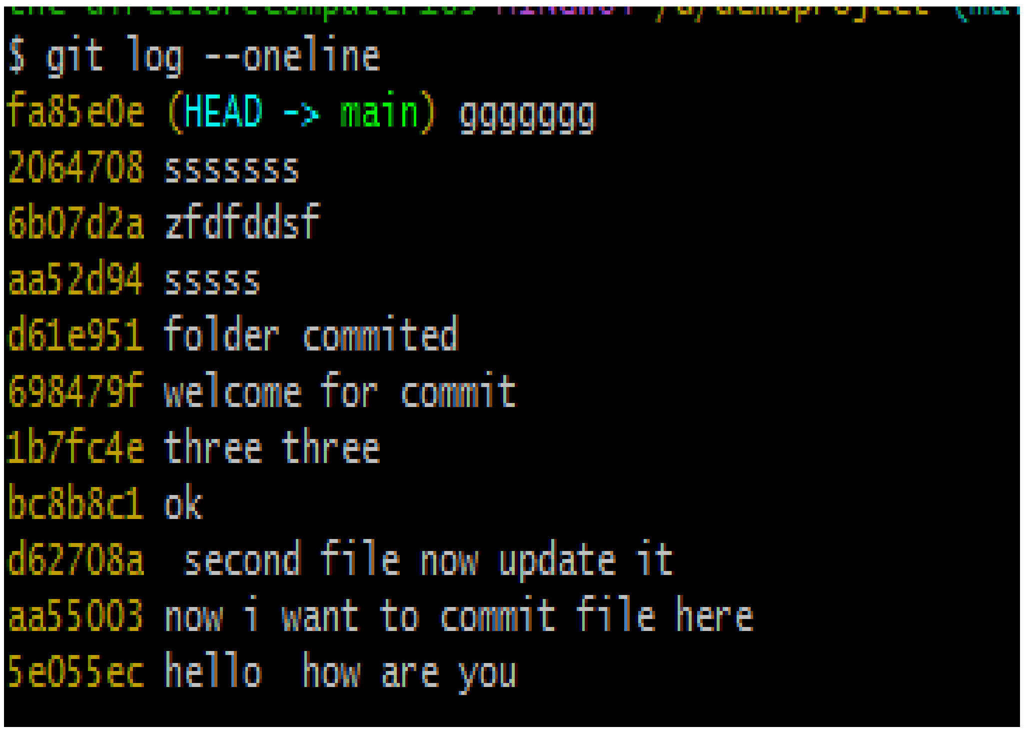
Viewing the Commit History :

$ git log --oneline

The oneline option is used to display the output as one commit per line. It also shows the output in brief like the first seven characters of the commit SHA and the commit message.

Viewing the Commit History :

$ git log --oneline



- One commit per line
- The first seven characters of the SHA
- The commit message

Viewing the Commit History :

$ git log --stat

The log command displays the files that have been modified. It also shows the number of lines and a summary line of the total records that have been updated.

Viewing the Commit History :

$ git log --stat

Generally, we can say that the stat option is used to display

- The modified files,
- The number of lines that have been added or removed.
- A summary line of the total number of records changed
- The lines that have been added or removed.

Viewing the Commit History :

$ git log --stat

```
$ git log --stat
commit fa85e0e69f48e93175671e0b98548d5ad416b429 (HEAD -> main)
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:11:35 2022 +0530

    ggggggg

 h.txt | 1 +
 1 file changed, 1 insertion(+)

commit 20647085cfd2a5c63e45276b8e16ed5b4cc282ac
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:08:02 2022 +0530

    sssssss

 hello.c | 1 -
 1 file changed, 1 deletion(-)

commit 6b07d2ac3d784eb16475829f5258f20debe6a26a
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:57:31 2022 +0530

    zfdfddsf

 th1.c | 1 +
 1 file changed, 1 insertion(+)

commit aa52d946b35bfd12558362425d6448ad95522d88
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:56:27 2022 +0530

    sssss
```

Viewing the Commit History :

$ git log --patch

The git log patch command displays the files that have been modified. It also shows the location of the added, removed, and updated lines.

Generally, we can say that the --patch flag is used to display:

-Modified files

-The location of the lines that you added or removed

-Specific changes that have been made.

Viewing the Commit History :

$ git log --patch

```
$ git log --patch
commit fa85e0e69f48e93175671e0b98548d5ad416b429 (HEAD -> main)
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:11:35 2022 +0530

    ggggggg

diff --git a/h.txt b/h.txt
new file mode 100644
index 0000000..c50bafc
--- /dev/null
+++ b/h.txt
@@ -0,0 +1 @@
+wel wel wel

commit 20647085cfd2a5c63e45276b8e16ed5b4cc282ac
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 14:08:02 2022 +0530

    sssssss

diff --git a/hello.c b/hello.c
deleted file mode 100644
index 1e06e51..0000000
--- a/hello.c
+++ /dev/null
@@ -1 +0,0 @@
-how are you

commit 6b07d2ac3d784eb16475829f5258f20debe6a26a
Author: priya sangita <sangita.phunde@rediffmail.com>
Date:   Thu Jun 16 13:57:31 2022 +0530

    zfdfddsf
```

Viewing the Commit History :

$ git log --graph

Git log command allows viewing your git log as a graph. To list the commits in the form of a graph, run the git log command with --graph option. It will run as follows:

$ git log --graph --oneline

Viewing the Commit History :

$ git log --graph

```
$ git log --graph
* commit fa85e0e69f48e93175671e0b98548d5ad416b429 (HEAD -> main)
| Author: priya sangita <sangita.phunde@rediffmail.com>
| Date:   Thu Jun 16 14:11:35 2022 +0530
|
|     ggggggg
|
* commit 20647085cfd2a5c63e45276b8e16ed5b4cc282ac
| Author: priya sangita <sangita.phunde@rediffmail.com>
| Date:   Thu Jun 16 14:08:02 2022 +0530
|
|     sssssss
|
* commit 6b07d2ac3d784eb16475829f5258f20debe6a26a
| Author: priya sangita <sangita.phunde@rediffmail.com>
| Date:   Thu Jun 16 13:57:31 2022 +0530
|
|     zfdfddsf
|
* commit aa52d946b35bfd12558362425d6448ad95522d88
| Author: priya sangita <sangita.phunde@rediffmail.com>
| Date:   Thu Jun 16 13:56:27 2022 +0530
|
|     sssss
|
```

Viewing the Commit History :

$ git log --graph --oneline

```
$ git log --graph --oneline
* fa85e0e (HEAD -> main) ggggggg
* 2064708 sssssss
* 6b07d2a zfdfddsf
* aa52d94 sssss
* d61e951 folder commited
* 698479f welcome for commit
* 1b7fc4e three three
* bc8b8c1 ok
* d62708a  second file now update it
* aa55003 now i want to commit file here
* 5e055ec hello  how are you
```

Common options to `git log`

Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the –stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

git restore :

git restore is used to restore or discard the uncommitted local changes of files.

Assume that you have done some changes in some files and then if you want to discard those local changes you can safely use **git restore**.

git restore <file_name>

git restore :

git restore <file_name>

Example:
git restore example.txt

git restore :

git restore one.txt two.txt   // Mention multiple files

git restore . // Discard all local changes

git restore *.rb // Wildcard option

2	2. Version Control-GIT 2.1. Introduction to GIT 2.2. What is Git 2.3. About Version Control System and Types 2.4. Difference between CVCS and DVCS 2.5. A short history of GIT 2.6. GIT Basics 2.7. GIT Command Line 2.8. Installing Git 2.9. Installing on Linux 2.10. Installing on Windows 2.11. Initial setup 2.12. Git Essentials 2.13. Creating repository 2.14. Cloning, check-in and committing 2.15. Fetch pull and remote 2.16. Branching 2.17. Creating the Branches, switching the branches, merging 2.18. The branches.	15	3

Git Branch

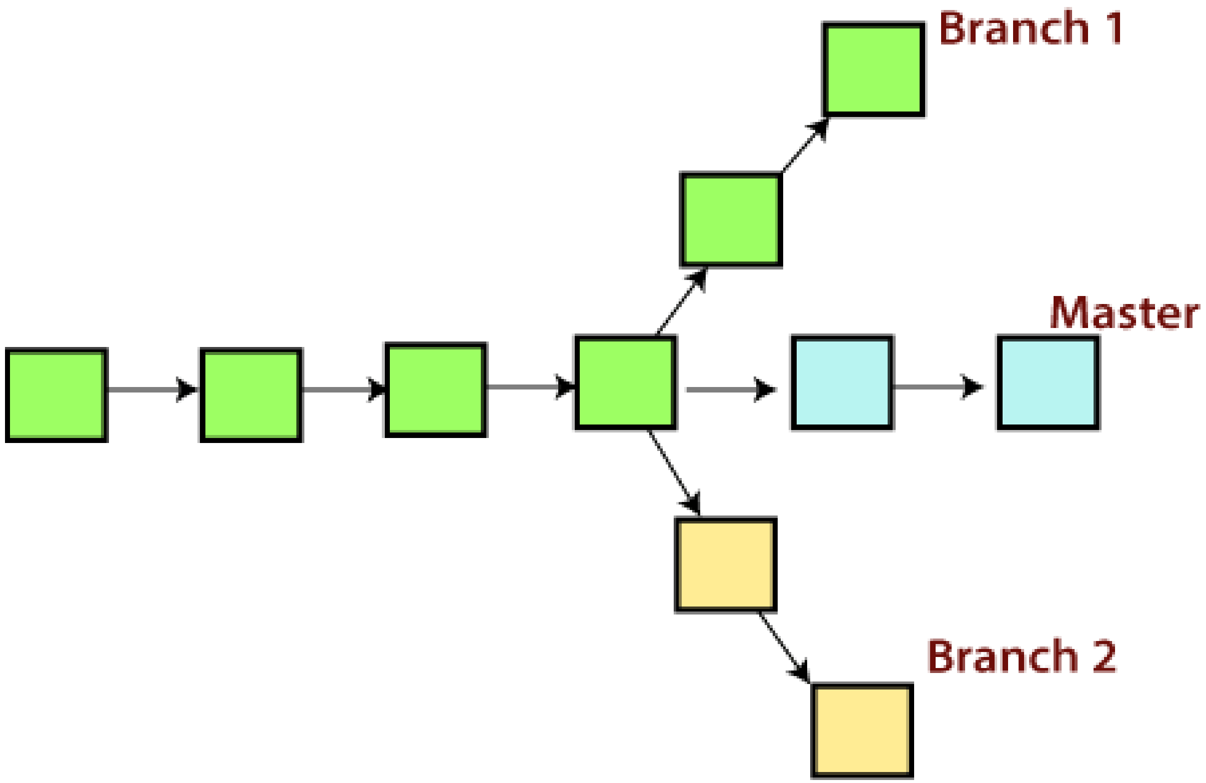
Git Branch:

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.

Git Branch:

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.

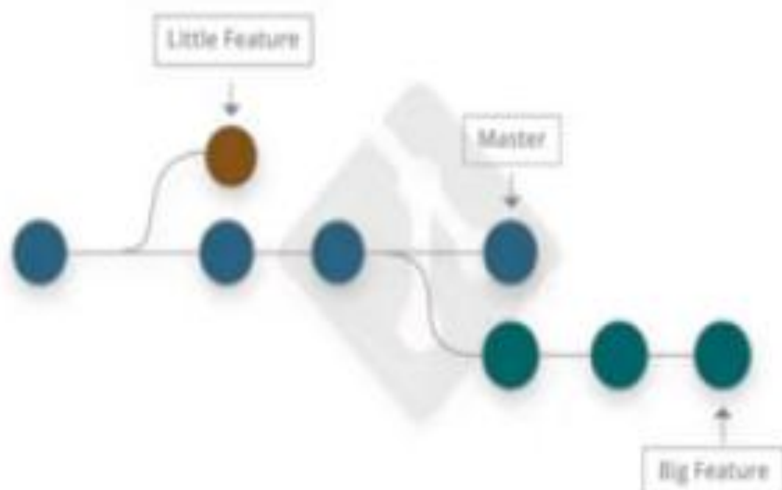
Git Branch:



WHAT ARE GIT BRANCHES?

- Pointer to a snapshot of your changes
- Independent line of development
- Default branch - Master

WHY DO WE NEED BRANCHES?



Two reasons:

- New features could break code
- Collaboration purposes

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.

Git 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.

Create Branch

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

Syntax:

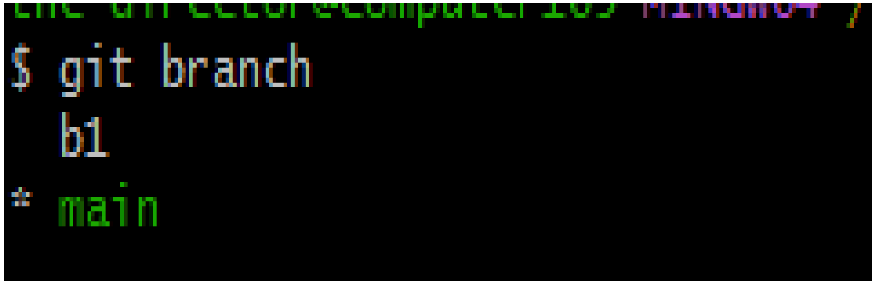
$ git branch <branch name>

```
$ git branch b1
```

List Branch

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

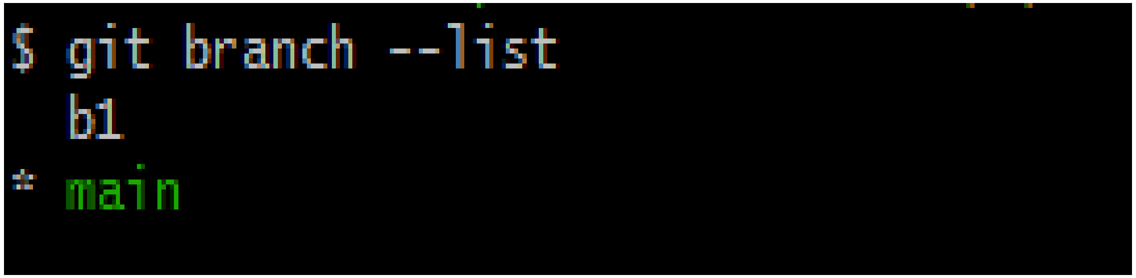
Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.



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.



List Branch

```
$ git branch
  b1
* main

the director@computer169 MINGW64 /d/demoproject (main)
$ git branch --list
  b1
* main
```

List Branch

```
$ git branch b2

the director@computer169 MINGW64 /d/demoproject (main)
$ git branch --list
  b1
  b2
* main
```

DELETING BRANCHES

- After merging branch into main code base, delete **local branch** without losing any history:

```
$ git branch -d new-local-repo
```

- If not merged, the above command will output an error message:

```
error: The branch 'new-local-repo' is not fully merged. If you are sure you want to delete it,
run 'git branch -D new-local-repo'.
```

- If you really want to delete the branch, you can use the capital -D flag:

```
$ git branch -D new-local-repo
```

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.

```
$ git branch -d<branch name>
```

Delete Branch :

```
$ git branch -d b1
Deleted branch b1 (was fa85e0e).
```

SWITCHING BRANCHES

- To switch to an existing branch, you run the 'git checkout' command

```
$ git checkout new-local-repo
```

- This moves HEAD to point to the new-local-repo branch

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 Branch :

```
$ git branch --list
  b2
* main

the director@computer169 MINGW64 /d/demoproject (main)
$ git checkout b2
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmoni
rue instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfi
se"
Switched to branch 'b2'
D       demo/wel.txt
A       hello.txt
A       one.c
D       th.c
D       th1.c
A       two.c
```

Working with Remotes

https://youtu.be/uaeKhfhYE0U

Working with Remotes :

To be able to collaborate on any Git project, you need to know how to manage your remote repositories.

Working with Remotes :

Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

Working with Remotes :

Collaborating with others involves managing these remote repositories and **pushing** and **pulling data** to and from them when you need to share work.

Working with Remotes :

Collaborating with others involves managing these remote repositories and **pushing** and **pulling data** to and from them when you need to share work.

Working with Remotes :

Remote repositories can be on your **local machine**.

Working with Remotes :

It is entirely possible that you can be working with a “remote” repository that is, in fact, on the same host you are. The word “remote” does not necessarily imply that the repository is somewhere else on the network or Internet, only that it is elsewhere.

Working with Remotes :

Working with such a remote repository would still involve all the standard pushing, pulling and fetching operations as with any other remote.

Showing Your Remotes :

To see which remote servers you have configured, you can run the git remote command.

It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see origin — that is the default name Git gives to the server you cloned from:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Enumerating objects: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0),
pack-reused 1857
Receiving objects: 100% (1857/1857), 334.06 KiB |
1.83 MiB/s, done.
Resolving deltas: 100% (837/837), done.
hint: core.useBuiltinFSMonitor=true is deprecated;
please set core.fsmonitor=true instead
hint: Disable this message with "git config advice
.useCoreFSMonitorConfig false"
```

$ git clone https://github.com/schacon/ticgit

$ cd ticgit

$ git remote

```
$ cd ticgit

the director@computer169 MINGW64 /d/demoproject/ticgit (master)
$ git remote
origin
```

You can also specify **-v,** which shows you the URLs that Git has stored for the short name to be used when reading and writing to that remote:

$ git remote -v

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Adding Remote Repositories :

We’ve mentioned and given some demonstrations of how the git clone command implicitly adds the origin remote for you. Here’s how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>

Adding Remote Repositories :

We’ve mentioned and given some demonstrations of how the git clone command implicitly adds the origin remote for you. Here’s how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>

Adding Remote Repositories :

We've mentioned and given some demonstrations of how the git clone command implicitly adds the origin remote for you. Here's how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

```
the director@computer169 MINGW64 /d/demoproject/ticgit (master)
$ git remote add pb https://github.com/paulboone/t
icgit

the director@computer169 MINGW64 /d/demoproject/ti
cgit (master)
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch
)
pb      https://github.com/paulboone/ticgit (push)
```

Now you can use the string pb on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run

```
$ git fetch pb
```

```
$ git fetch pb
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fs
rue instead
hint: Disable this message with "git config advice.useCoreFSMonitorC
se"
remote: Enumerating objects: 43, done.
remote: Counting objects: 100% (22/22), done.
remote: Total 43 (delta 22), reused 22 (delta 22), pack-reused 21
Unpacking objects: 100% (43/43), 5.99 KiB | 2.00 KiB/s, done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Fetching and Pulling from Your Remotes :

$ git fetch <remote>

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

Fetching and Pulling from Your Remotes :

If you clone a repository, the command automatically adds that remote repository under the name “origin”.

So, **git fetch origin** fetches any new work that has been pushed to that server since you cloned (or last fetched from) it.

Fetching and Pulling from Your Remotes :

It’s important to note that the **git fetch command** only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on.

Fetching and Pulling from Your Remotes :

If your current branch is set up to track a remote branch (see the next section and Git Branching for more information), you can use the git pull command to automatically fetch and then merge that remote branch into your current branch.

Fetching and Pulling from Your Remotes :

This may be an easier or more comfortable workflow for you; and by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from.

Fetching and Pulling from Your Remotes :

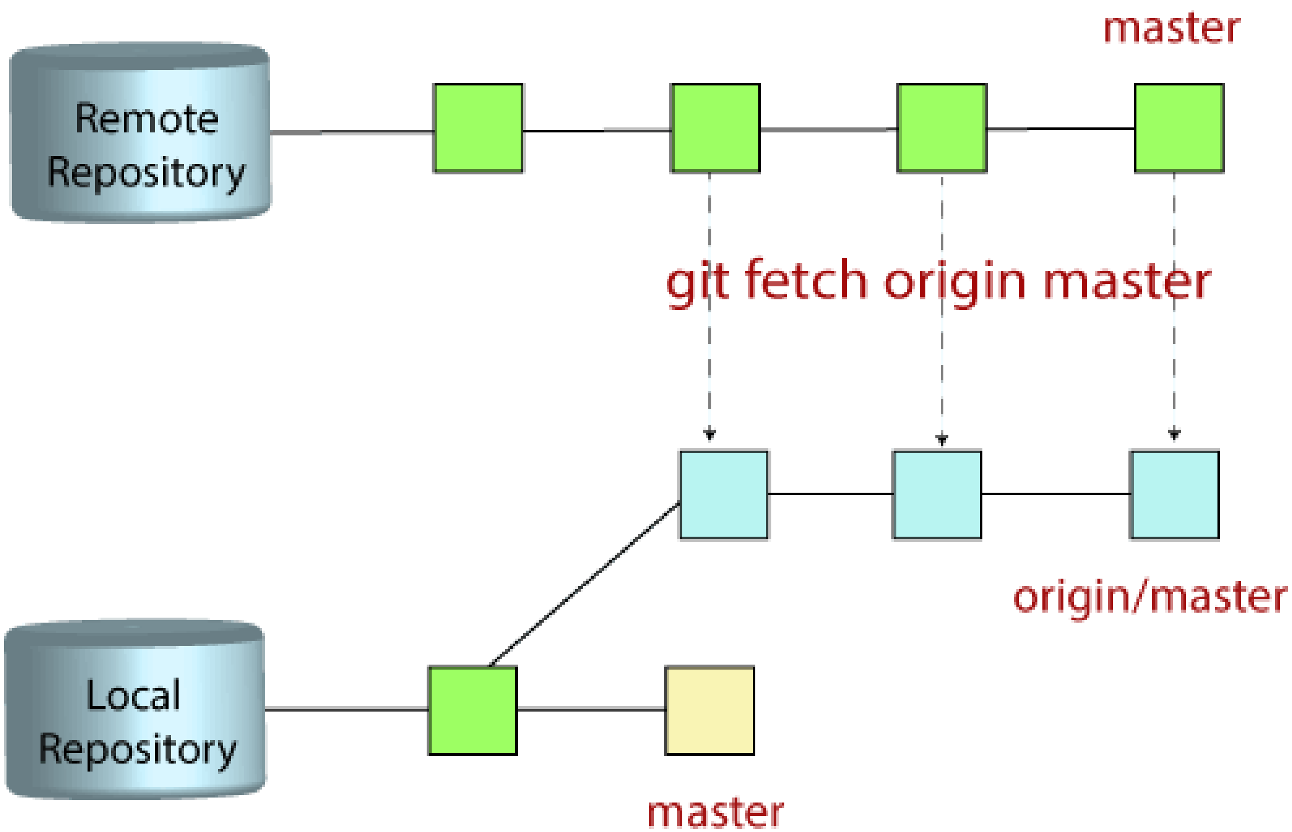
Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

Git Fetch

Git Fetch :

Git **"fetch"** Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.

Git Fetch :


master
Remote
Repository
git fetch origin master
origin/master
Local
Repository
master

Git Fetch :

The **"git fetch" command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

Git Fetch :

```
$ git fetch <remote>

$ git fetch origin

$ git fetch pb
```

```
the director@computer169 MINGW64 /d/demoproject/ticgit (master)
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch)
pb      https://github.com/paulboone/ticgit (push)
```

```
$ git fetch origin
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=t
stead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig fal

the director@computer169 MINGW64 /d/demoproject/ticgit (master)
$ git fetch pb
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=t
stead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig fal

the director@computer169 MINGW64 /d/demoproject/ticgit (master)
$
```

Pushing to Your Remotes

Git Push :

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple:

git push <remote> <branch>.

If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

Pushing to Your Remotes

Git Push :

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push

- echo "# demo" >> README.md
- git init
- git add README.md
- git commit -m "first commit"
- git branch -M main
- git remote add origin

  https://github.com/sangitaphunde/demo.git
- git@github.com:sangitaphunde/imsdemo.git
- **git push -u origin main**

Inspecting a Remote :

If you want to see more information about a particular remote, you can use the git remote show <remote> command. If you run this command with a particular shortname, such as origin, you get something like this:

$ git remote show origin

Renaming and Removing Remotes :

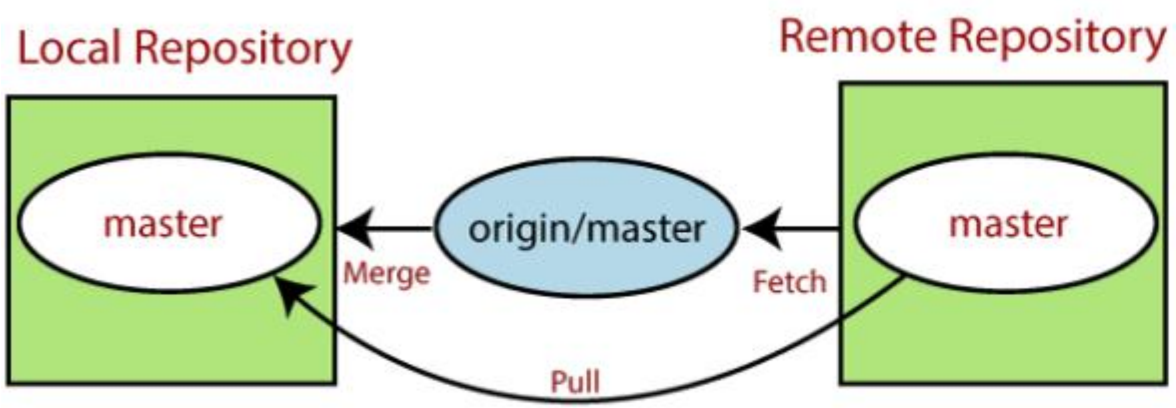
$ git remote rename pb paul
$ git remote
origin
paul

Renaming and Removing Remotes :

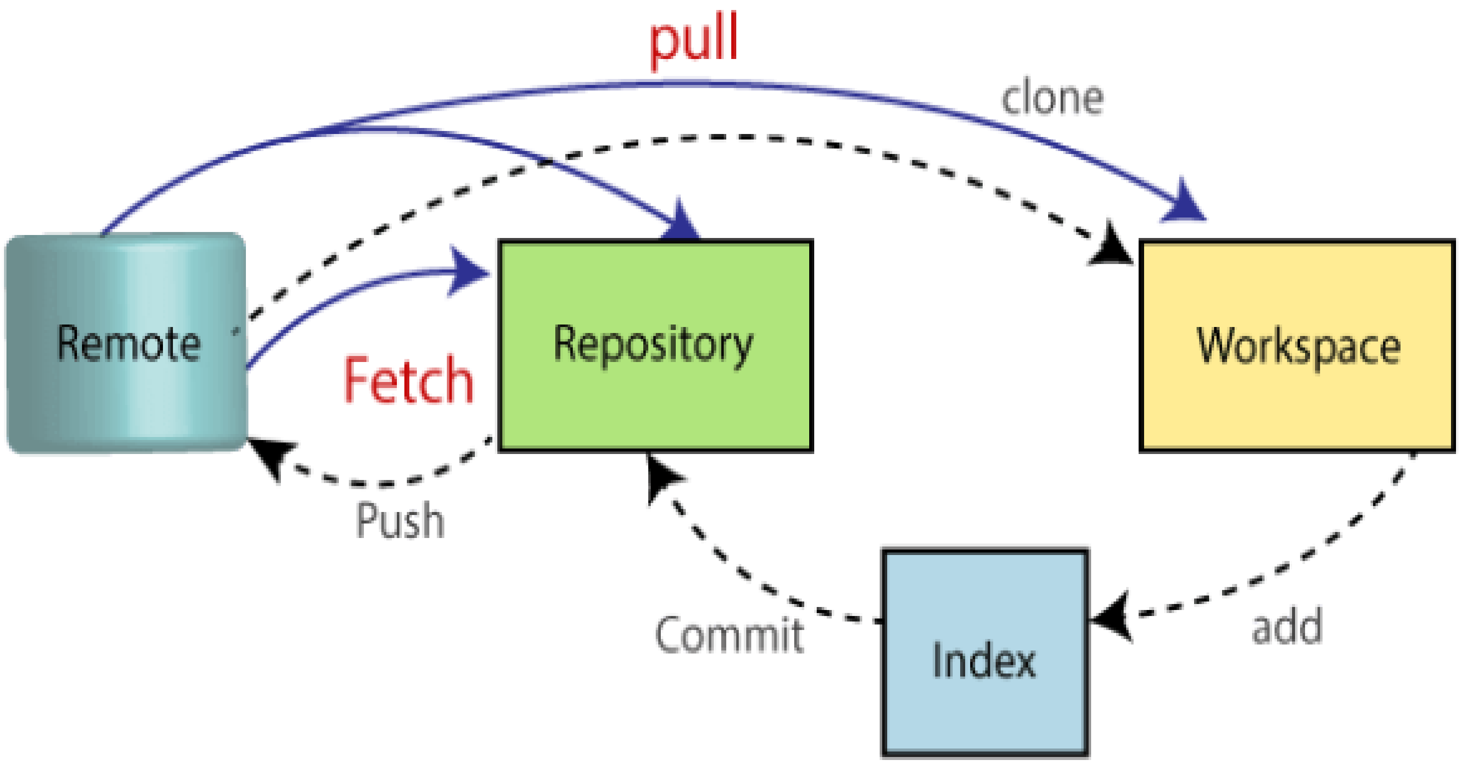
```
$ git remote remove paul
$ git remote
origin
```

Git Pull / Pull Request :

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repo



Git Pull / Pull Request :



Git Pull / Pull Request :

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

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo (master)
$ git pull https://github.com/ImDwivedi1/GitExample2.git
remote: Enumerating objects: 38, done.
remote: Counting objects: 100% (38/38), done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 38 (delta 13), reused 19 (delta 7), pack-reused 0
Unpacking objects: 100% (38/38), done.
From https://github.com/ImDwivedi1/GitExample2
 * branch            HEAD       -> FETCH_HEAD

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo (master)
$
```

Git Pull / Pull Request :

$ git pull <options><remote>/<branchname>

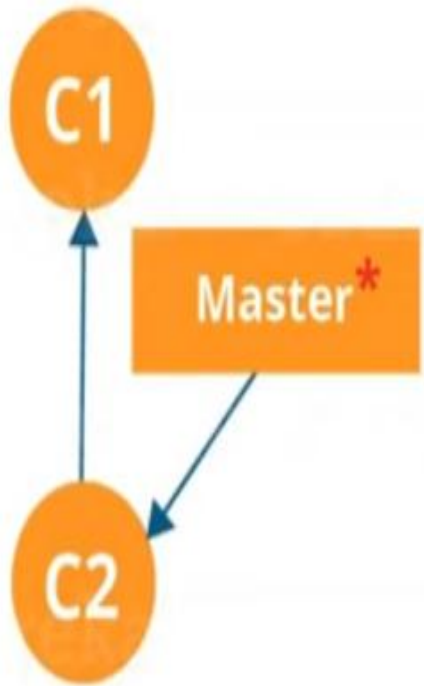
$ git pull origin master

$ git checkout -b hotfix

Above command create new branch hotfix and switched to that branch also.

Git Branch

git branch



git branch	Checkout your current branch
git merge	Integrate branches together
git checkout	Used for switching branches

```
$ git checkout master

$ git merge hotfix
```

Finally merge the hotfix branch back into your master branch to deploy to production.

$git diff

```
$ git diff
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonito
rue instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig
se"
warning: LF will be replaced by CRLF in a.txt.
The file will have its original line endings in your working directory
diff --git a/a.txt b/a.txt
index ccc3e7b..8240350 100644
--- a/a.txt
+++ b/a.txt
@@ -1 +1,2 @@
 aaaaa
+welcome
```

- **Shows the difference between worked area and stage area files**

Moving Files :

$ git mv file_from file_to

$ git mv README.md README

$ git status

On branch master

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

Changes to be committed:

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

renamed: README.md -> README

```
$ mv hello.c hhee.c

the director@computer169 MINGW64 /d/demoproject (main)
$ git status
hint: core.useBuiltinFSMonitor=true is deprecated;please set core.fsmonitor=true instead
hint: Disable this message with "git config advice.useCoreFSMonitorConfig false"
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hello.c
        new file:   one.c
        new file:   th.c
        new file:   th1.c
        new file:   two.c

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    hello.c

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

2	2. Version Control-GIT 2.1. Introduction to GIT 2.2. What is Git 2.3. About Version Control System and Types 2.4. Difference between CVCS and DVCS 2.5. A short history of GIT 2.6. GIT Basics 2.7. GIT Command Line 2.8. Installing Git 2.9. Installing on Linux 2.10. Installing on Windows 2.11. Initial setup 2.12. Git Essentials 2.13. Creating repository 2.14. Cloning, check-in and committing 2.15. Fetch pull and remote 2.16. Branching 2.17. Creating the Branches, switching the branches, merging 2.18. The branches.	15	3

About SSH key generation

About SSH key generation :

If you want to use a hardware security key to authenticate to GitHub, you must generate a new **SSH key for your hardware security key.** You must connect your hardware security key to your computer when you authenticate with the key pair.

Generating a new SSH key

1] Open Git Bash.

2] Paste the text below, substituting in your GitHub email address.

$ ssh-keygen -t ed25519 -C *your_email@example.com*

$ ssh-keygen -t ed25519 -C “*sangita.phunde@rediffmail.com*"

Generating a new SSH key

3] When you're prompted to "Enter a file in which to save the key," press Enter. This accepts the default file location.

> Enter a file in which to save the key

(/c/Users/*you*/.ssh/id_*algorithm*):*[Press enter]*

Generating a new SSH key

4] At the prompt, type a secure passphrase. For more information, see "Working with SSH key passphrases.“

>Enter passphrase (empty for no passphrase): *[Type a passphrase]*

> Enter same passphrase again: *[Type passphrase again]*

Adding a new SSH key to your GitHub account

$ clip < ~/.ssh/id_ed25519.pub # Copies the contents of the id_ed25519.pub file to your clipboard

Adding a new SSH key to your GitHub account

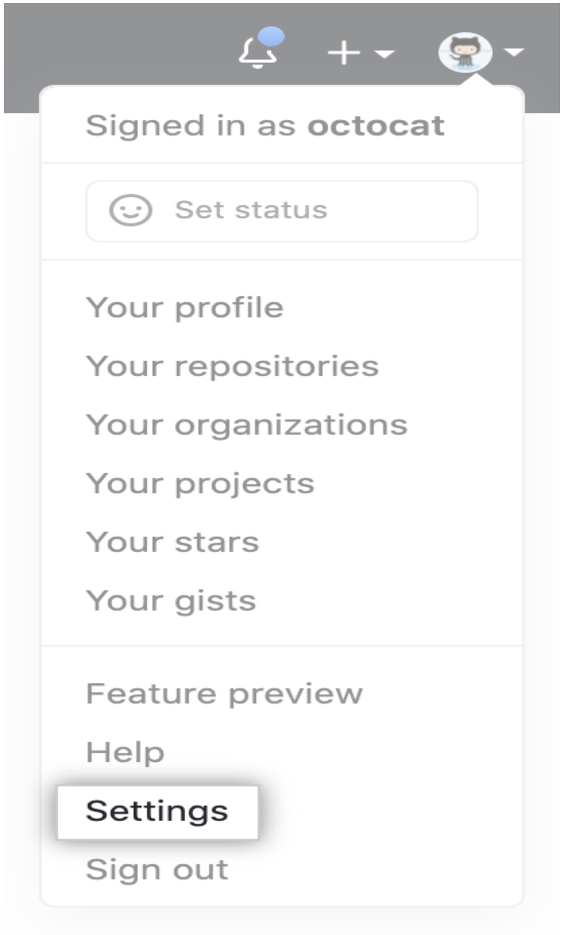
1] Copy the SSH public key to your clipboard.

```
$ clip < ~/.ssh/id_ed25519.pub
```

Copies the contents of the id_ed25519.pub file to your clipboard

Adding a new SSH key to your GitHub account

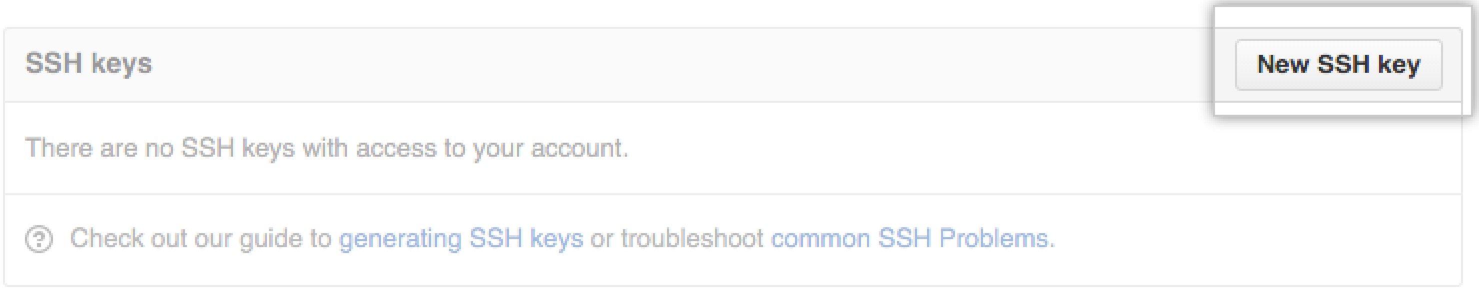
2] In the upper-right corner of any page, click your profile photo, then click **Settings**.



Adding a new SSH key to your GitHub account

- In the "Access" section of the sidebar, click **SSH and GPG keys.**

- Click **New SSH key** or **Add SSH key.**

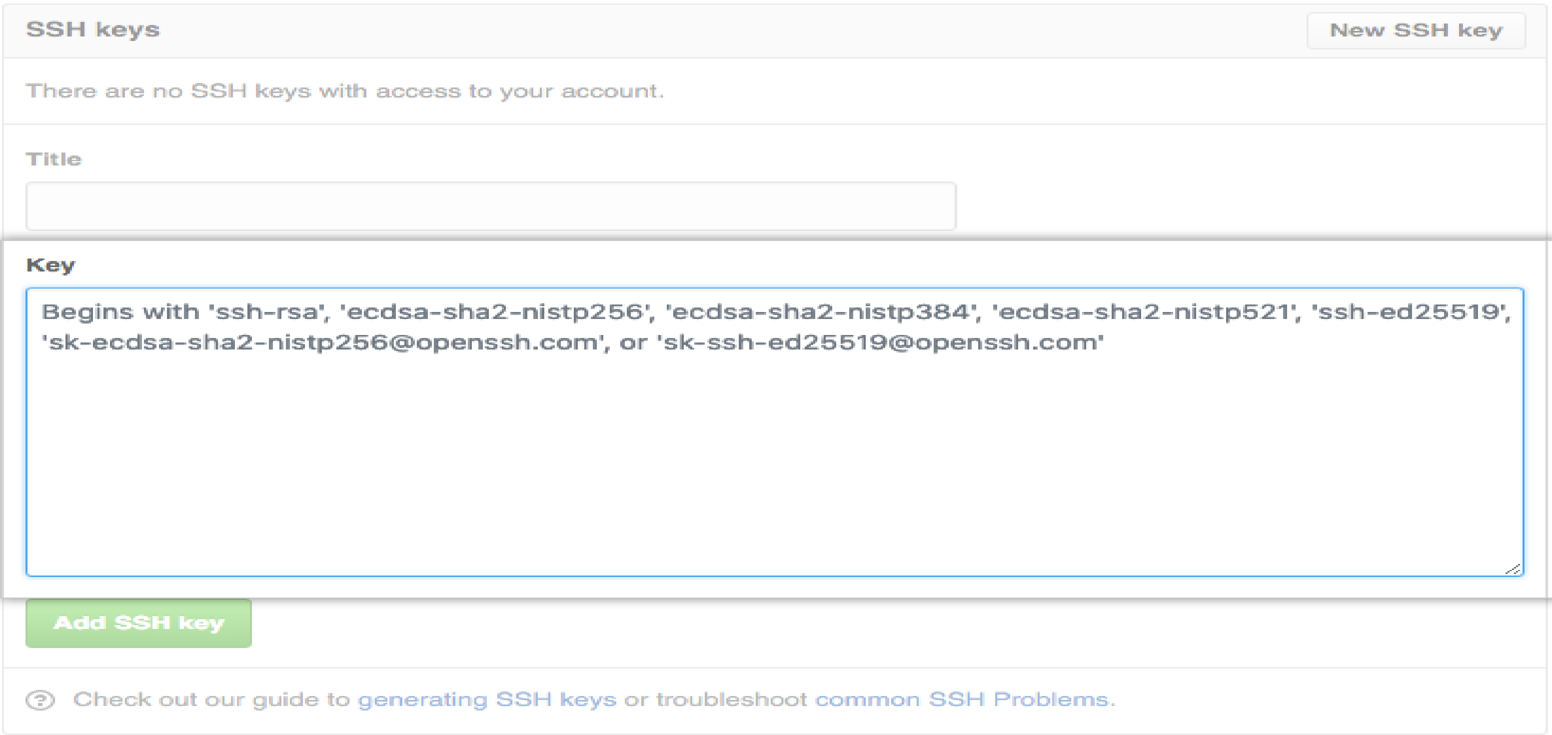


Adding a new SSH key to your GitHub account

- In the "Title" field, add a descriptive label for the new key. For example, if you're using a personal Mac, you might call this key "Personal MacBook Air".

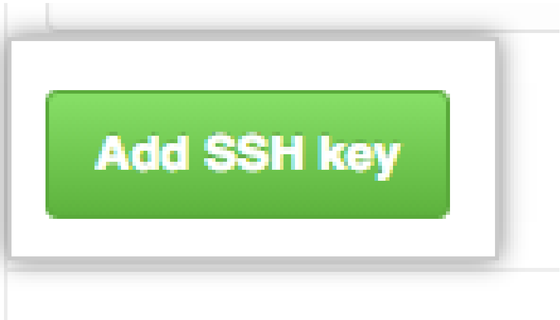
Adding a new SSH key to your GitHub account

- Paste your key into the "Key" field.



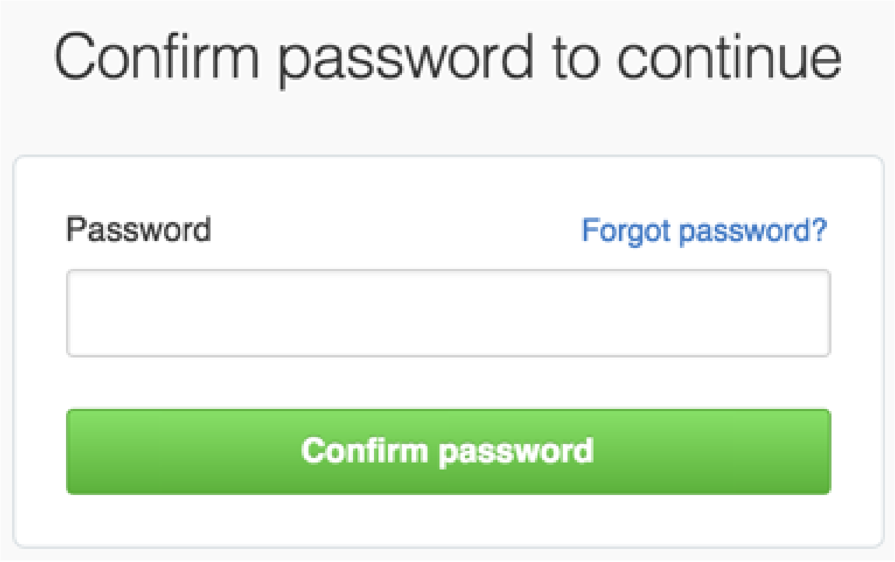
Adding a new SSH key to your GitHub account

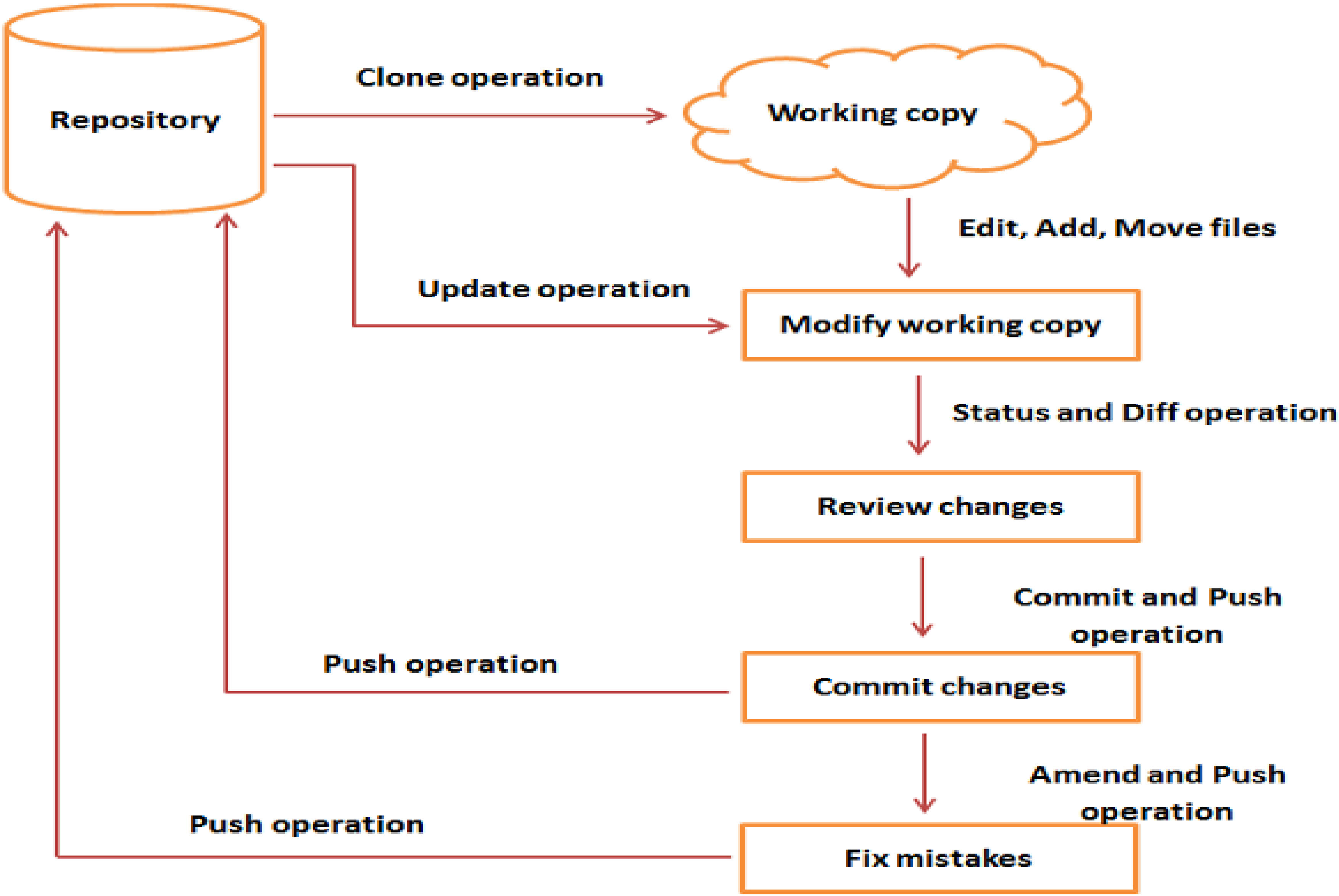
- Click **Add SSH key**.



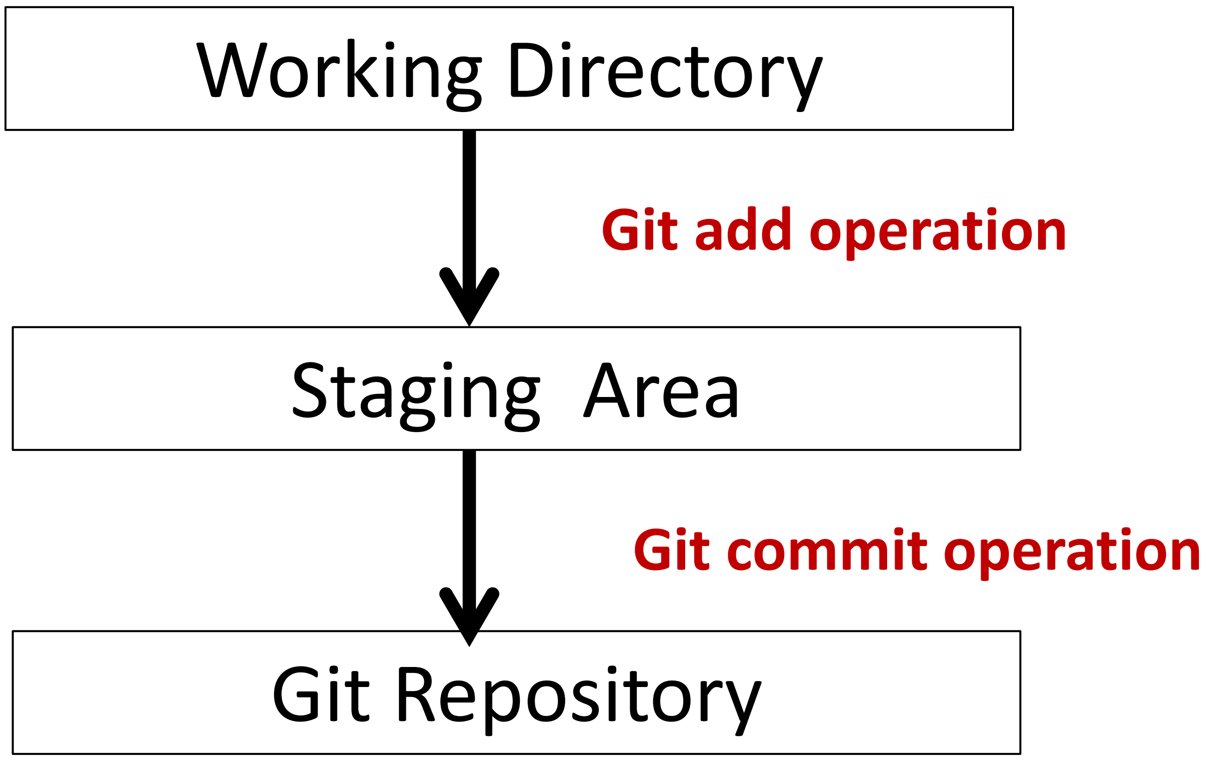
Adding a new SSH key to your GitHub account

- If prompted, confirm your GitHub password.





Git - Life Cycle


Working Directory
Git add operation
Staging Area
Git commit operation
Git Repository

DVCS

Ignoring Files :

Not everything in the working directory should be tracked by Git. There are certain files (config, passwords, bad code) that are generally left untracked by authors or developers.

Ignoring Files :

Those files (or directories) are listed in a simple file called “.gitignore”. Notice the period before “gitignore”, it is important. To ignore files, create a file named .gitignore and list the files or folders to ignore in it.