1. **Git Introduction**
2. **Git Installation**
3. **Git architecture**

**Git Introduction**

We’re going to cover Version Control Systems (VCSs) and Git basics — no technical stuff, just what Git is, why it came about in a land full of VCSs, what sets it apart, and why so many people are using it. We’ll explain how to download Git and set it up for the first time if you don’t already have it on your system.

1. What is Version control Systems
2. What are the types of Version Control Systems
3. What is Git
4. Features/Advantages of Git
5. Disadvantages of Git

**What is Version control System?**

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over a period of time so that you can recall specific versions later.

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

**What are the types of Version Control Systems?**

1. Local version control
2. Centralized Version Control Systems
3. Distributed Version Control Systems

**Local version control**

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error-prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to. To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control. One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

[RCS - GNU Project - Free Software Foundation](https://www.gnu.org/software/rcs/)

**Centralized Version Control Systems:**

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, 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 several clients that check out files from that central place. For many years, this has been the standard for version control.

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.

However, 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. 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 happens to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

SVN => Subversion was created by [CollabNet](https://en.wikipedia.org/wiki/CollabNet) Inc. in 2000, and is now a top-level Apache project being built and used by a global community of contributors.

**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. 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 a full backup of all the data. Furthermore, 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. 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 a 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 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 (See Git Branching).

**What is Git?**

So, what is Git in a nutshell? This is an important section to absorb because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion, or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git’s user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

**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).

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 takes a picture of what all your files look like at that moment and stores a reference to that snapshot. 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.

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 filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We’ll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Git Branching.

**Nearly Every Operation Is Local**

Most operations in Git need only locally 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, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

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. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that it is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your local copy, remember?) until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can’t do much when you aren’t connected to the server; in Subversion and CVS, you can edit files, but you can’t commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

**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. The mechanism that Git uses for this checksumming is called an 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. An SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. Git stores everything in its database not by file name but by the hash value of its contents.

**What Git is Lacking:**

Features GIT (open source) is lacking

1. Lacks UI
2. LDAP Integration- Authentication
3. RBAC- Role-based Access Control Authorization
4. Auditing – Who logged in when
5. High Availability – Clustering
6. Notifications
7. Third-party tool integrations
8. Backup & restore.

**Repository**

The repository is a folder in which we have the sourcecode + version history

1. **Git Installation**

After Installation First-Time Git Setup

1. Git name
2. Git email
3. Git editor preference

**Your Identity**

The first thing you should do when you install Git is to set your user’s name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

Configure tooling Configure user information for all local repositories

**$ git config --global user.name "[name]"**

Sets the name you want to be attached to your committed transactions

**$ git config --global user. email "[email address]"**

Sets the email you want to be attached to your commit transactions

**Git default Editor settings**

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor

**$ git config --global core. editor emacs - for MAC**

**$ git config – global core. emacs - for MAC**

**$ git config -global core. editor “nano” - Linux**

**$ git config –global core. editor “’c:/Program files/Notepad++/notepad++.exe’ -multi-list -notabbar -session -noPlugin”**

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

**Text Color**

**$ git config --global color. UI auto**

Enables helpful colorization of the command line output

**Settings of End of Line conversion**

**Windows**  **Linux/Mac**

Carriage return linefeed

Linefeed

**Core.autocrlf ()**

**LF into CRLF (Linefeed into windows CRLF from Linux or Mac to windows)**

$ git config – global core. autocrlf true

**CRLF into LF (windows CRLF into Linefeed (While pushing the code into Linux))**

$ git config –global core. autocrlf input

**How to unset the user.name and email**

git config --global --unset credential.helper

git config --unset credential.helper

CR = Carriage Return ( \r , 0x0D in hexadecimal, 13 in decimal) — moves the cursor to the beginning of the line without advancing to the next line.

LF = Line Feed ( \n , 0x0A in hexadecimal, 10 in decimal) — moves the cursor down to the next line without returning to the beginning of the line.

A carriage return means moving the cursor to the beginning of the line.

The code is \r.

A line feed means moving one line forward to next line. The code is \n.

Windows editors often still use the combination of both as \r\n in text files.

Unix uses mostly only the \n.

The separation comes from typewriter times, when you turned the wheel to move the paper to change the line and moved the carriage to restart typing on the beginning of a line. This was two steps.

**Git architecture**

**The Three States**

**Pay attention now** — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: modified, staged, and committed:

• **Working tree/Working Area/Modified** means that you have changed the file but have not committed it to your database yet.

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 a disk for you to use or modify.

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

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.

• **Committed/ Git Directory** means that the data is safely stored in your local database.

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.

**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.

To set main as the default branch name do:

**$ git config --global init. defaultBranch main**

**Checking Your Settings**

**$ git config --list --show-origin**

You can view all of your settings and where they are coming from using:

**$ git config --list**

**Workspace:**

Local checkout of your code. Also called “working copy” or just “checkout”

**Index:**

A staging area for file changes to commit. Before you commit (or check-in) files, you need to first add them to the index. This is also called “current directory cache”, “staging area”,” cache”, or “staged files”.

**Local Repository:**

A directory named. git that contains all your necessary repository files-- a Git repository skeleton. Typical branches: main, master, feature-X, bugfix-y. The local repository has exactly the same features and functionality as any other git repository.

**Upstream repository:**

A repository of your code to share and collaborate with other developers. It’s hosted on some the Internet or a remote, eg. Github. The default name is origin. Typical branches here: master, main, shared-feature-x, release-y. Also called “remote repository”, or just ‘remote’.

**Git stash:**

A place to hide modifications while you work on something else:

This Happens form workspace to stash and vice-versa:

Stash push [<msg>]

Stash pop

Stash apply [<stash>]

**Problems before VCS?**

1. Versioning was manual
2. Team collaboration was a time-consuming and hectic task
3. No easy access to previous versions
4. Multiple versions took a lot of space

we will go over basic Git usage — how to use Git in the 80% of cases you’ll encounter most often. After reading this chapter, you should be able to clone a repository, see what has happened in the history of the project, modify files, and contribute changes. If the book spontaneously combusts at this point, you should already be pretty useful wielding Git in the time it takes you to go pick up another copy.

Git Basics If you can read only one chapter to get going with Git, this is it.

This chapter covers every basic command you need to do the vast majority of the things you’ll eventually spend your time doing with Git.

By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes.

We’ll also show you how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

**Getting and Creating Projects**

There are two ways to get a Git repository.

One is to create a new one in an existing directory.

The other is to copy it from an existing repository on the network or elsewhere and

1. git init
2. git clone

We are going to see a first way to create a git repository.

The second one we will see on Collaboration session.

**Basic Snapshotting**

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

1. git init
2. git add
3. git commit
4. git status
5. git log
6. git stage
7. git restore –staged filename
8. git diff
9. git difftool
10. git reset -- soft hash
11. git rm
12. git mv
13. git clean
14. git blame
15. git stash
16. git stash list/drop/save/apply/pop
17. git show
18. git shortlog
19. git describe
20. git ls-files tell us all the files in the staging area