Git for Non Technical Readers

Adrian Kosmaczewski

2021-10-08

If you are in the business of software, sooner or later you will hear people talking about Git, GitHub, or GitLab. What are they? To explain that, we must learn what Git is first.

- What is Git?
- Managing File Versions
- Software
- Version Control
- Terminology
- History
  - First Generation: No Networking Support, and Single-user
  - Second Generation: Centralized, and Multi-user or "Concurrent"
  - Third Generation: Distributed
- GitLab, GitHub, BitBucket…
- Other Tools for Git

What is 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.

There are several things in the definition above that make absolutely no sense to people that aren't that much into computers.

- "Free and open source": that's probably the only part that is actually easy to understand; it means that the tools are freely available to all, without cost or obligations.
- "Distributed": there's an idea of locality, of space here.
- "Version control": clearly something to do with those "versions" that all software packages mention in their websites.

So the complicated parts here are "distributed" and "version control". Let's start with the last one first.

Managing File Versions

All of us have experienced the typical collaboration project, where many people work together on the same Word or Photoshop file. First the file is called `draft-1.psd`, then comes `final-edited-by-john.docx`, then it becomes `draft-for-comments-14.xlsx`, and next thing you know, contributions are scattered among tens of files and nobody actually knows what the "actual" final document is.

Of course these days there are online collaboration tools such as Google Docs, Confluence, or Office 365 that allow several people to edit the same file at once, and you can see their cursors on the screen as they write. Still, even with this capability, we might end up with several versions of files laying around.

Thankfully many of these tools offer a "history" function showing a list of versions, and their respective author and creation date. Well, Git does precisely something like this.

Software

In the world of software, source code is the basic unit of collaboration. Source code is just text written in some language, which is then used to "build" the final "thing" that is installed in a PC or in a server, like Word, or this very website. For a single software project, you can have hundreds of different files, sometimes even hundreds of thousands of them.

Needless to say, keeping track of it all is very, very complicated without help.

But early on, developers realized that, when working in a software package consisting of many files, it is very rare that two developers are editing the same line of the same file at once.

Hence, it makes sense to allow people to work together, since the actual chance of collision is really, really low.

And if developers need to work together in the same problem, they might want to do it on the computer of one of them, sharing the keyboard and exchanging ideas (or, these days, over Zoom) which removes any chance of conflict. This technique, by the way, is usually referred to as "Extreme Programming" or "XP".

Version Control

A "Version Control System" provides teams with the ability to keep every change made, to every file, in every folder, of every project, forever. Every change in the project is stored along with the following information:

- **Who**: the name and e-mail of the person that made the commit
- **When**: the date and time of the change
- **What**: the changes themselves

About the "what", here's an important point: to be efficient, version control systems do not store a complete file every time that it changes; but only the lines that changed. It seems complicated (it is, to a certain degree) but the core concept is, precisely, this.

And Git is, essentially, just another Version Control System.

Terminology

Every project in a version control system is called a "repository". A repository contains all the files in a project, stored within folders. The version control system allows users to see more than just the files; it allows to see the "history" of each files, and also to "blame" (ugly word, indeed) and see who wrote any line of code in any file of the repository.

Remember the "history" feature I mentioned before? Well, here it is.

This tracking allows developers to understand the evolution of the project with the finest possible grain.

When developers start working in a project, they "check out" a copy of the project on their own computer. Through the network, they receive the files, and can work on them: add features, solve bugs, add documentation, even execute tests, or performing some management task like counting the lines of code in the project.

Whenever developers make a modification that works (that is, a new feature or a fixed bug) they "commit" the change. This adds a new entry in the history of the project.

Every so often, developers can "tag" a particular commit, so that they can find faster a previous version. For example "v1.0" would be a very useful and valid tag. You can tag projects any way you want, with words or numbers. Anything goes, really.

History

Others have written about the history of version control systems, so I'm just going to link to them here:

- Eric Sink.
- Lynda
- Redgate

For simplicity, I like to group the history of version control systems in three generations.

First Generation: No Networking Support, and Single-user

The first version control systems were designed during the 1970s; the need to collaborate on source code clearly dates from the early times of computers.

Two important early systems are:

- SCSS released in 1973, written in SNOBOL for the IBM System/370 running OS/360 (closed source until 2005)
- GNU RCS started in 1982, still maintained! (free, open source)

These systems did not cope well with many users editing the same file, which was a major drawback.

Second Generation: Centralized, and Multi-user or "Concurrent"

During the 1980s, computers starting getting connected in networks, which meant that teams could keep a centralized repository, and would check out and commit changes from their own computers.

Many very influential systems were built around this model.

- CVS started in 1986 (free, open source) => the "C" stands for "Concurrent"
- Subversion started in 2000 (free, open source) as a "better CVS" with transactional commits.
- Microsoft SourceSafe (commercial, closed source, early 90s)
- Microsoft Team Foundation Server (Microsoft, today "Azure DevOps Server") (commercial, closed source, 2010)
- IBM Rational ClearCase (commercial, closed source, mid-90s) => great reputation, very high cost
- SourceGear Vault (commercial, closed source, late 90s) marketed as a safer alternative to SourceSafe, targeting the same demographic.
- Perforce (commercial, closed source, 1995) still very much used inside Google.

Centralized systems use a simple architecture: a server stores a database with all the source code, while clients connect and get the "current copy", usually the latest.

These systems had a terrible Achiles' Heel: if the server was corrupted, all the project was lost. This was unfortunately very common with Microsoft SourceSafe, who earned a terrible reputation back in the 1990s. Of course this requires a backup strategy.

Third Generation: Distributed

- BitKeeper (free, open source, 2000) open source since 2016, commercial before.
- GNU arch (free, open source, 2001) now deprecated.

- GNU Bazaar (free, open source, 2005)
- Git (free, open source, 2005) created by the Linux team
- Mercurial (free, open source, 2005) which is losing popularity lately while Git grows
- Fossil (free, open source)

Git is the *de facto* standard today. All the others are niche, used in very small segments of the market. From that perspective, they are almost nonexistent.

With Git, there is no server anymore; every member of a software development team has a full copy (a "clone") of all the history of all the project. Clones can share changes with one another without distinction. Of course this makes it very, very hard for a malfunction to bring down the whole project; there's plenty of backups in every computer that cloned the repository.

Speaking about terminology, when joining a new project, developers must first "clone" its repository. Then they "commit" changes locally, and after that, they "push" those changes to the source. Every so often they "pull" changes, to get all the changes done by their colleagues.

This is exactly the model used at many companies today.

As a side note, it is important to point out that CVS allowed users to import RCS projects; Subversion could import CVS repositories. And finally, Git allows to import Subversion repositories. This is exactly the path that many projects followed since the 1980s.

GitLab, GitHub, BitBucket...

Even if Git is distributed, developers usually keep a "designated clone" at the center of their collaboration. This requires a centralized server where the "master clone" resides.

GitHub was among the first services to provide such a functionality, and one of the first to do it based on Git. Just like with any other SaaS, users can sign up, create a repository, and then clone it and push changes to it.

A GitHub project is nothing else than a Git project; this means that users can "clone" the project away from GitHub and host it into their own comptuters, or even into a competitor of GitHub, and life goes on.

Given the success of GitHub (eventually bought by Microsoft) came BitBucket (from Atlassian) and finally GitLab, which can be self-hosted. This means that it can be installed "on premises", which makes it very popular with enterprises. Another common self-hosted system of the same kind is Gitea.

All of these options provide many features, complementing the mere fact of just storing repositories:

- Web-based interface accessible from any operating system and web browser.

- A terminal (CLI) or mobile application.
- Repository creation, grouping, management.
- History review and exploration for repositories.
- Continuous integration and deployment (CI/CD).
- Integrated wiki (like Confluence).
- Ticket management (like Jira).
- Project management (milestones, etc).
- Integration with other tools, such as Kubernetes or Terraform.
- User rights management.
- Analytics & statistics.

Other Tools for Git

Git being an open source tool, there are lots and lots of tools that you can use to browse, review, manage, and interact with Git repositories. Among the most popular we can find the following:

- Atlassian SourceTree for Windows and macOS.
- Tower for Windows and macOS.
- TortoiseGit for Windows
- Visual Studio Code for Windows, Mac, & Linux, is one of a myriad of text editors with built-in Git support. It has plenty of extensions, among which I can recommend the following two related to Git:
  - Git Graph.
  - Annotator