Introduction

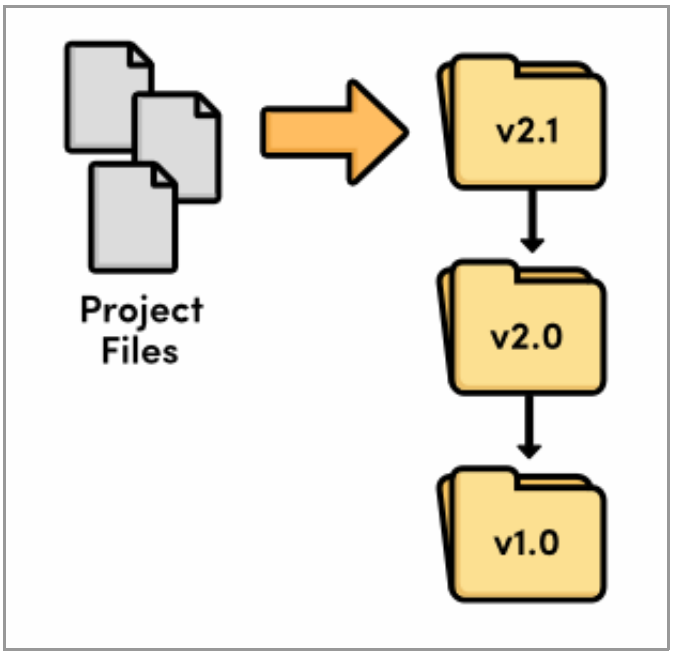
Git is a version control system (VCS) created for a single task: managing changes to your files. It lets you track every change a software project goes through, as well as where those changes came from. This makes Git an essential tool for managing large projects, but it can also open up a vast array of possibilities for your personal workflow.

A Brief History of Revision Control

We'll talk more about the core philosophy behind Git in a moment, but first, let's step through the evolution of version control systems in general.

Files and Folders

Before the advent of revision control software, there were only files and folders. The only way to track revisions of a project was to copy the entire project and give it a new name. Just think about how many times you've saved a "backup" called `my-term-paper-2.doc`. This is the simplest form of version control.

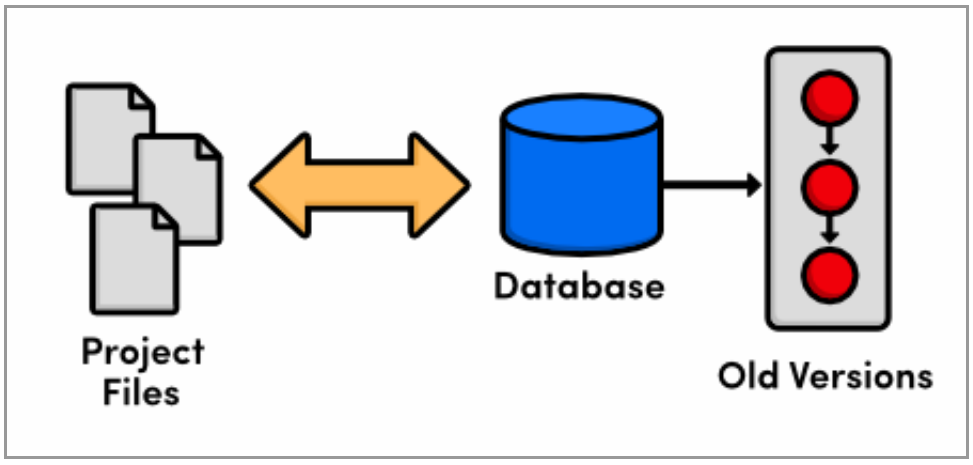


Revision control with files and folders

But, it's easy to see how copying files from folder to folder could prove disastrous for software developers. What happens if you mis-label a folder? Or if you overwrite the wrong file? How would you even know that you lost an important piece of code? It didn't take long for software developers to realize they needed something more reliable.

Local VCS

So, developers began writing utility programs dedicated to managing file revisions. Instead of keeping old versions as independent files, these new VCSs stored them in a database. When you needed to look at an old version, you used the VCS instead of accessing the file directly. That way, you would only have a single "checked out" copy of the project at any given time, eliminating the possibility of mixing up or losing revisions.

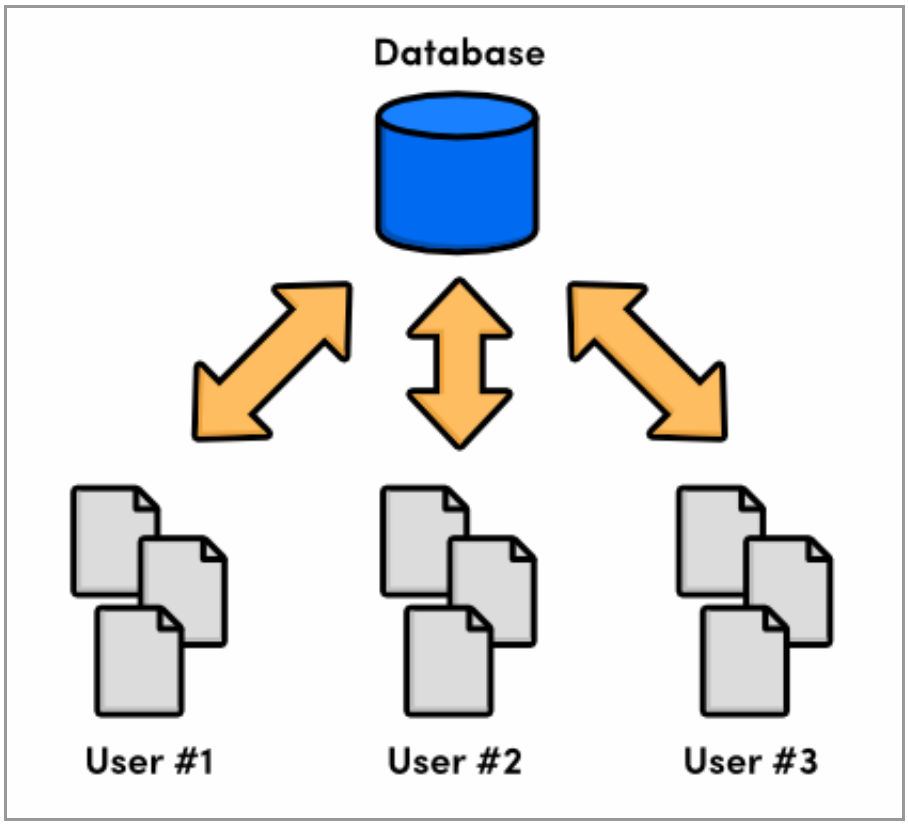


Local version control

At this point, versioning only took place on the developer's *local* computer—there was no way to efficiently share code amongst several programmers.

Centralized VCS

Enter the centralized version control system (CVCS). Instead of storing project history on the developer's hard disk, these new CVCS programs stored everything on a server. Developers checked out files and saved them back into the project over a network. This setup let several programmers collaborate on a project by giving them a single point of entry.



Centralized version control

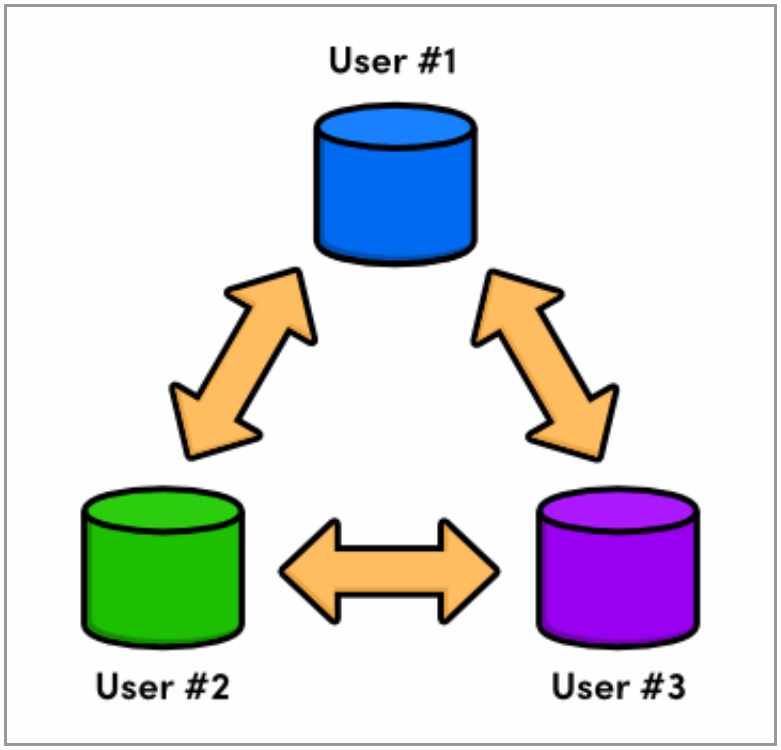
While a big improvement on local VCS, centralized systems presented a new set of problems: how do multiple users work on the same files at the same time? Just imagine a scenario where two people fix the same bug and try to commit their updates to the central server. Whose changes should be accepted?

CVCSs addressed this issue by preventing users from overriding others' work. If two changes conflicted, someone had to manually go in and merge the differences. This solution worked for projects with relatively few updates (which meant relatively few

conflicts), but proved cumbersome for projects with dozens of active contributors submitting several updates everyday: development couldn't continue until all merge conflicts were resolved and made available to the entire development team.

Distributed VCS

The next generation of revision control programs shifted away from the idea of a single centralized repository, opting instead to give every developer their own *local* copy of the entire project. The resulting *distributed* network of repositories let each developer work in isolation, much like a local VCS—but now the conflict resolution problem of CVCS had a much more elegant solution.



Distributed version control

Since there was no longer a central repository, everyone could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience. In addition, distributed version control systems (DVCS) focused on efficient management for separate branches of development, which made it much easier to share code, merge conflicts, and experiment with new ideas.

The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. And, since each user had a complete copy of the project, the risk of a server crash, a corrupted repository, or any other type of data loss was much lower than that of their CVCS predecessors.

The Birth of Git

And so, we arrive at Git, a distributed version control system created to manage the Linux kernel. In 2005, the Linux community lost their free license to the BitKeeper software, a commercial DVCS that they had been using since 2002. In response, Linus Torvalds advocated the development of a new open-source DVCS as a replacement. This was the birth of Git.

As a source code manager for the entire Linux kernel, Git had several unique constraints, including:

- Reliability
- Efficient management of large projects
- Support for distributed development
- Support for non-linear development

While other DVCSs did exist at the time (e.g., GNU's Arch or David Roundy's Darcs), none of them could satisfy this combination of features. Driven by these goals, Git has been under active development for several years and now enjoys a great deal of stability, popularity, and community involvement.

Git originated as a command-line program, but a variety of visual interfaces have been released over the years. Graphical tools mask some of the complexity behind Git and often make it easier to visualize the state of a repository, but they still require a solid foundation in distributed version control. With this in mind, we'll be sticking to the command-line interface, which is still the most common way to interact with Git.

Installation

The upcoming modules will explore Git's features by applying commands to real-world scenarios. But first, you'll need a working Git installation to experiment with. Downloads for all supported platforms are available via the official Git website.

For Windows users, this will install a special command shell called *Git Bash*. You should be using this shell instead of the native command prompt to run Git commands. OS X and Linux users can access Git from a normal shell. To test your installation, open a new command prompt and run `git --version`. It should output something like `git version 1.7.10.2 (Apple Git-33)`.

Get Ready!

Remember that *Ry's Git Tutorial* is designed to *demonstrate* Git's feature set, not just give you a superficial overview of the most common commands. To get the most out of this tutorial, it's important to actually execute the commands you're reading about. So, make sure you're sitting in front of a computer, and let's get to it!

Continue to *The Basics* ›