VERSION CONTROL WITH GIT

Excerpt from:
Learning Web Design, 5e

by Jennifer Robbins

Author's Note: *This article first appeared in Learning Web Design, 5e, released in 2018. It was removed from the 6th edition, but I have made it available here as supplemental material. Keep in mind that some details may be out of date.*

If you've done work on a computer, you've probably used some sort of system for keeping track of the versions of your work. You might have come up with a system of naming drafts until you get to the "final" version (and the "final-final" version, and the "final-final-no-really" version, and so on). You might take advantage of macOS's Time Machine to save versions that you can go back to in an emergency. Or you might have used one of the professional version control systems that have been employed by teams for decades.

The king of version control systems (VCS) for web development is a robust program called Git (*git-scm.com*). At this point, knowing your way around Git is a requirement if you are working on a team and is a good skill to have even for your own projects.

In this section, I'll introduce you to the terminology and mental models that will make it easier to get started with Git. Teaching all the ins and outs of how to configure and use Git from the command line is a job for another book and online tutorials (I list a few at the end of the section), but I wish someone had explained the difference between a "branch" and a "fork" to me when I was starting out, so that's what I'll do for you.

We'll begin with a basic distinction: Git is the version control program that you run on your computer; GitHub (*github.com*) is a service that hosts Git projects, either free or for a fee. You interact with GitHub by using Git, either from the command line, with the user interface on the GitHub website, or using a standalone application that offers a GUI interface for Git commands. This was not obvious to me at first, and I want it to be clear to you from the get-go.

IN THIS ARTICLE

FUN FACT

Git was created by Linus Torvalds, the creator of the Linux operating system, when he needed a way to allow an enormous community to contribute to the Linux project.

GitHub and services like it (see Note) are mainly web-based wrappers around Git, offering features like issue tracking, a code review tool, and a web UI for browsing files and history. They are convenient, but keep in mind that you can also set up Git on your own server and share it with your team members with no third-party service like GitHub involved at all.

NOTE

Beanstalk (beanstalkapp.com), GitLab (gitlab.com), and Bitbucket (bitbucket.org) are other Git hosting services aimed at enterprise-scale projects. GitLab has a free option for public projects, similar to GitHub, and because it is open source, you can host it yourself. Search the web for "Git hosting services" to find up-to-

WHY USE GIT

There are several advantages to making Git (and GitHub) part of your workflow. First, you can easily roll back to an earlier version of your project if problems show up down the line. Because every change you make is logged and described, it helps you determine at which point things might have gone wrong.

Git also makes it easy to collaborate on a shared code source. You may tightly collaborate with one or more developers on a private project, merging all of your changes into a primary copy. As an added benefit, the sharing process is a way to get an extra set of eyes on your work before it is incorporated. You may also encourage loose collaboration on a public project by welcoming contributions of people you don't even know in a way that is safe and managed. Git is a favorite tool for this type of collaboration on all sorts of open source projects.

Git is a favorite tool for collaboration on open source projects.

Getting up to speed with GitHub in particular is important because it's what everyone is using. If your project is public (accessible to anyone), the hosting is free. For private and commercial projects, GitHub charges a fee for hosting. In addition to hosting projects, they provide collaboration tools such as issue tracking. You may have already found that some of the links to tools I mentioned in this book go to GitHub repositories. I want you to know what you can do when you get there.

HOW GIT WORKS

Git keeps a copy of every revision of your files and folders as you go along, with every change (called a commit) logged in with a unique ID (generated by Git), a message (written by you) describing the change, and other metadata. All of those versions and the commit log are stored in a repository, often referred to as a "repo."

Once you have Git installed on your computer, every time you create a new repository or clone an existing one, Git adds a directory and files representing the repo's metadata alongside other files in the project's folder. Once the Git repository is initialized, you can commit changes and take advantage of the "time machine" feature if you need to get back to an earlier version. In this way, Git is a good tool for a solo workflow.

More likely you'll be working with a team of other folks on a project. In that case, a hub model is used in which there is an official repository on a central server that each team member makes a local copy of to work on. Each team member works on their own machine, committing to their local repo, and at logical intervals, uploads their work back to the central repository.

That's what makes Git a distributed version control system compared to other systems, like SVN, that require you to commit every change directly to the server. With Git, you can work locally and offline.

The first part of mastering Git is mastering its vocabulary. Let's run through some of the terminology that will come in handy when you're learning Git and the GitHub service. FIGURE A is a simplified diagram that should help you visualize how the parts fit together.

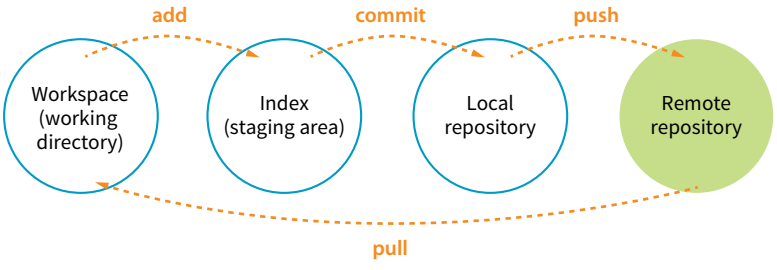

FIGURE A. Visualization of Git structure.

Working Directory

The working directory is the directory of files on your computer in which you do your actual work. Your working copy of a file is the one that you can make changes to, or to put it another way, it's the file you can open from the hard drive by using Finder or My Computer.

Repository

Your local Git repository lives alongside the files in your working directory. It contains copies, or snapshots, of all the files in a single project at every step in its development, although these are kept hidden. It also contains the metadata stored with each change. There may also be a central repository for the project that lives on a remote server like GitHub.

Commit

A commit is the smallest unit of Git interaction and the bulk of what you'll do with Git. Git uses "commit" as a verb and a noun. You may save your working document frequently as you work, but you commit (v.) a change when you want to deliberately add that version to the repository. Usually, you

Git Visualization Resources

Need more help picturing how all these pieces and commands work together? Try these visualization resources:

- The Git Cheatsheet from NDP Software provides a thorough interactive mapping of how various Git commands correspond to the workspace and local and remote repositories. It's worth checking out at *ndpsoftware.com/git-cheatsheet.html#loc=workspace*.
- A Visual Git Reference (*marklodato.github.io/visual-git-guide/index-en.html*) is a collection of diagrams that demonstrate most common Git commands.
- "Understanding the GitHub Flow" (*guides.github.com/introduction/flow/*) explains a typical workflow in GitHub.

commit at a logical pause in the workflow—for example, when you've fixed a bug or finished changing a set of styles.

When you commit, Git records the state of all the project files and assigns metadata to the change, including the username, email, date and time, a unique multidigit ID number (see the "Hashes" sidebar), and a message that describes the change. These stored records are referred to as commits (n.). A commit is like a snapshot of your entire repository—every file it contains—at the moment in time you made the commit.

Commits are additive, so even when you delete a file, Git adds a commit to the stack. The list of commits is available for your perusal at any time. On GitHub, use the History button to see the list of commits for a file or folder.

The level of granularity in commits allows you to view the repository (project) at any state it's ever been at, ever. You *never* lose work, even as you proceed further and further. It's a great safety net. Indirectly this also means that there's nothing you can do with Git that you can't undo—you can't get yourself into an impossible situation.

Hashes

The unique ID that Git generates for each commit is technically called a SHA-1 hash, more affectionately known in the developer world as simply a hash. It is a 40-character string written in hexadecimal (0–9 and A–F are used), so the odds of having a duplicate hash are astronomical. It's common to use short hashes on projects instead of the full 40 characters. For example, on GitHub, short hashes are seven characters long, and you'll see them in places like a project's Commits page. Even with just seven characters, the chances of collision are tiny.

Staging

Before you can commit a change, you first have to make Git aware of the file (or to track it, to use the proper term). This is called staging the file, accomplished by adding it to Git. In the command line, it's `git add` *filename*, but other tools may provide an Add button to stage files. This creates a local index of files that you intend to commit to your local repository but haven't been committed yet. It is worth noting that you need to "add" any file that you've changed, not just new files, before committing them. Staging as a concept may take a little while to get used to at first because it isn't especially intuitive.

Branch

A branch is a sequential series of commits, also sometimes referred to as a stack of commits. The most recent commit on any given branch is the head (see Note). You can also think of a branch as a thread of development. Projects usually have a primary or default branch, typically (although not necessarily) called main, which is the official version of the project. To work on a branch, you need to have it checked out.

When working in a branch, at any point you can start a new branch to do a little work without affecting the source branch. You might start a new branch to experiment with a new feature, or to do some debugging, or to play around with presentation. Branches are often used for small, specific tasks like that, but you can create a new branch for any purpose you want.

For example, if you are working on "main," but want to fix a bug, you can create a new branch off main and give the branch a new descriptive name, like "bugfix." You can think of the bugfix branch as a copy of main at the point

NOTE

There are exceptions, as it is possible to reorder commits; however, it is almost always true that the head commit is also the most recent.

at which bugfix was created (FIGURE B), although that's not exactly what is happening under the hood.

To work on the bugfix branch, you first need to check it out (**`git checkout bugfix`**), and then you can go about your business of making changes, saving them, adding them to Git, and committing them. Eventually, the new branch ends up with a commit history that is different from the source branch.

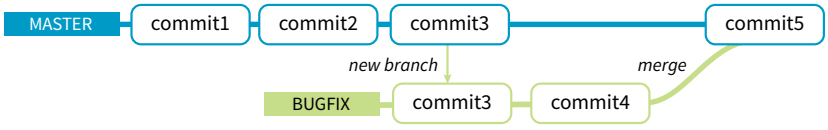

FIGURE B. Creating and merging a new branch.

When you are done working on your new branch, you can merge the changes you made back into the source branch and delete the branch. If you don't like what's happening with the new branch, delete it without merging, and no one's the wiser.

Merging

Merging is Git's killer feature for sharing code. You can merge commits from one branch into another (such as all of the commits on a feature branch into main) or you might merge different versions of the same branch that are on different computers. According to the Git documentation, merging "incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch." Put another way, Git sees merging as "joining two histories together," so it useful to think of merging happening at the commit level.

Git attempts to merge each commit, one by one, into the target branch. If only one branch has changed, the other branch can simply fast-forward to catch up with the changes. If both branches have commits that are not in the other branch—that is, if both branches have changes—Git walks through each of those commits and, on a line-by-line basis, attempts to merge the differences. Git actually changes the code inside files for you automatically so you don't have to hunt for what's changed.

However, if Git finds conflicts, such as two different changes made to the same line of code, it gives you a report of the conflicts instead of trying to change the code itself. Conflicts are pointed out in the source files between ======= and <<<<<<< characters (FIGURE C). When conflicts arise, a real person needs to read through the list and manually edit the file by keeping the intended change and deleting the other. Once the conflicts are resolved, the files need to be added and committed again.

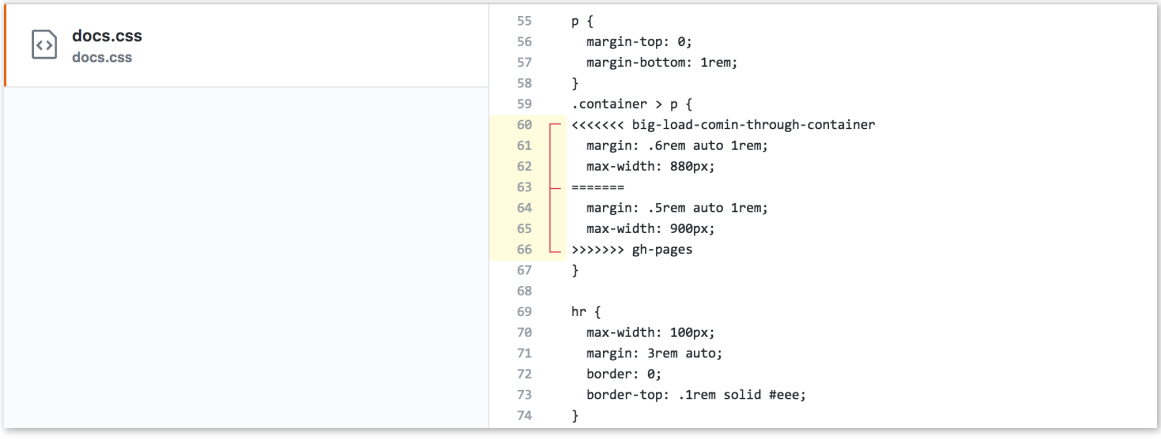

FIGURE C. GitHub conflict report.

Remotes

All of the features we've looked at so far (commits, branches, merges) can be done on your local computer, but it is far more common to use Git with one or more remote repositories. The remote repo could be on another computer within your organization, but it is likely to be hosted on a remote server like GitHub. Coordinating with a remote repository opens up a few other key Git features.

Clone

Cloning is making an exact replica of a repository and everything it contains. It's common to clone a repo from a remote server to your own computer, but it is also possible to clone to another directory locally. If you are getting started on an existing project, making a clone of project's repo is a logical first step.

Push/pull

If you are working with a remote repository, you will no doubt need to upload and download your changes to the server. The process of moving data from your local repository to a remote repository is known as pushing. When you push commits to the remote, they are automatically merged with the current version on the server. To update your local version with the version that is on the server, you pull it, which retrieves the metadata about the changes and applies the changes to your working files. You can think of pushing and pulling as the remote version of merging.

It is a best practice to pull the remote main repo frequently to keep your own copy up-to-date. That helps eliminate conflicts, particularly if there are a lot

of other people working on the code. Many GUI Git tools provide a Sync button that pulls and pushes in one go.

Fork

You may hear talk of "forking" a repo on GitHub. Forking makes a copy of a GitHub repository to your GitHub account so you have your own copy to play around with. Having the repo in your account is not the same as having a working copy on your computer, so once you've forked it, you need to clone (copy) it to your own computer (FIGURE D).

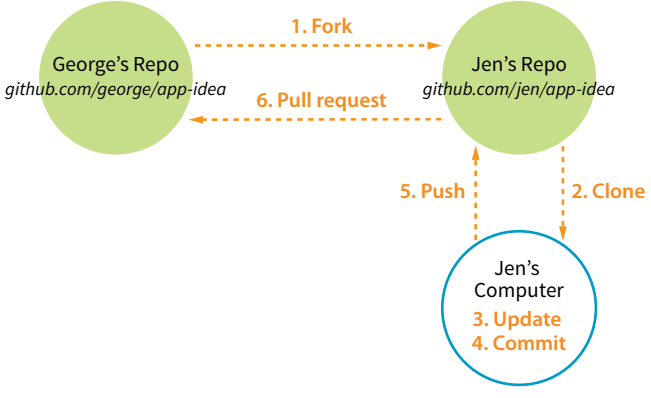

FIGURE D. Once you fork a repository on GitHub, you need to clone it to get a local working copy. (Based on a diagram by Kevin Markham.)

People fork projects for all sorts of reasons (see Note). You might just want to have a look under the hood. You may want to iterate and turn it into something new. You may want to contribute to that project in the form of pull requests. In any scenario, forking is a safeguard for repository owners so they can make the project available to the public while also controlling what gets merged back into it.

Pull Request

It is important to keep in mind that your forked copy is no longer directly connected to the original repository it was forked from. You will not be able to push to the original. If you come up with something you think is valuable to the original project, you can do what is called a pull request—that is, asking the owner to pull your changes into the original "main" branch.

You can also do a pull request for a repo that you have access to, not just one that you've forked. For example, if you've made a branch off the main project branch, you can do a pull request to get your team to review what you've done and give you feedback before merging your changes back in. In fact, pull requests may be used earlier in the process to start a discussion about a possible feature.

GIT TIP

Always *pull* before you *push* to avoid conflicts.

NOTE

Forking is most often used for contributing to an open source project. For commercial or personal projects, you generally commit directly to the repository shared by your team.

GIT TOOLS AND RESOURCES

Most Git users will tell you that the best way to use Git is with the command line. As David Demaree says in his book *Git for Humans*, "Git's command-line interface is its native tongue." He recommends typing commands and seeing what happens as the best way to learn Git. The downside of the command line, of course, is that you need to learn all the Git commands and perhaps also tackle the command-line interface hurdle itself. The following resources will help get you up to speed:

- *Pro Git* by Scott Chacon and Ben Straub (Apress) is available free online (*git-scm.com/book/en/v2*).
- *Git for Humans* by David Demaree (A Book Apart) is a great place to start learning Git via the command line (or however you intend to use it!).
- "Git Cheat Sheet" from GitHub is a list of the most common commands (*education.github.com/git-cheat-sheet-education.pdf*).
- The Git Reference Manual on the official Git site provides a thorough listing of commands and features (*git-scm.com/docs*).

There are also several graphical Git applications available for those who prefer icons, buttons, and menus for interacting with their repositories, and there's no shame in it. I know many developers who use a graphical app and Terminal side by-side, choosing the tool that most easily allows them to do the task they need to do. If you feel more comfortable getting started with a graphical Git tool, I recommend the following:

- GitHub Desktop (from GitHub) is free and available for Mac and Windows (*github.com/apps/desktop*).
- Git Tower 2 (Mac and Windows) costs money, but it is more powerful and offers a thoughtfully designed interface, including visualizations of branches and merges (*www.git-tower.com*).

Many code editors have built-in Git support or Git/GitHub plug-ins as well.

If you go to the GitHub.com site, they do a good job of walking you through the setup process with easy-to-follow tutorials. You can set up an account and gain some basic GitHub skills in a matter of minutes. Their online documentation is top-notch, and they even have a YouTube channel with a video playlist of tutorials aimed at beginners (*www.youtube.com/@GitHub*).

And speaking of GitHub, for a good introduction to the ins and outs of the GitHub interface, I recommend the book *Introducing GitHub: A Non-Technical Guide* by Brent Beer (O'Reilly).

When you are ready to get started using Git for version control, you'll find all the support you need.