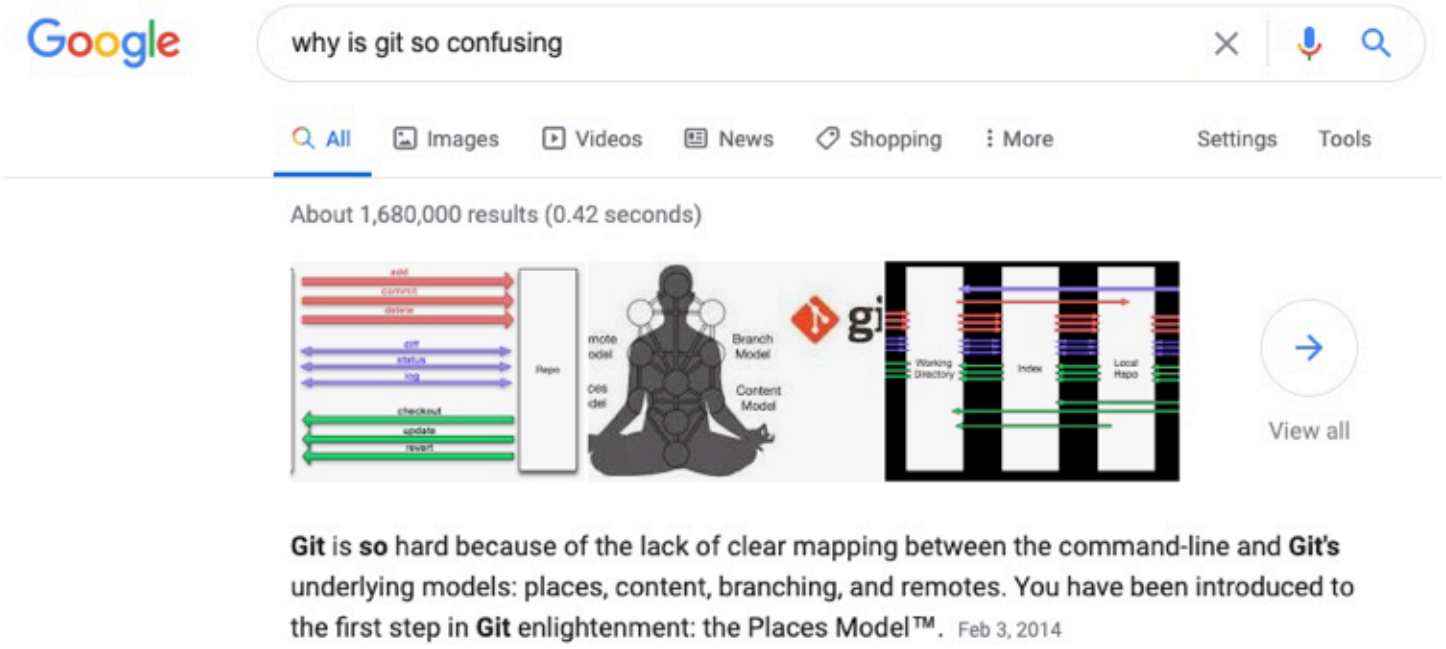
Git/Github Introduction

5 minute read

March 4, 2021 by Bradley Voytek



Version control for software used to be a *huge* pain. Imagine co-writing a document with someone. We use Google docs now, which lets multiple people edit the document at once, and it saves a history of each edit. You can open up a browser and look at that history and if you don't like an edit (or whole set of edits) you can "roll back" (**git reset**) to that previous version.

But it used to be that we'd email a Word doc to someone when we wanted their edits, and then we'd wait for them to email their edits back. And that sucks and means multiple people can't easily work on the same docs at the same time.

So the modern solution is Git and GitHub.

Git is the software that is used for tracking edits to files. Every time a tracked file is edited and saved (**git commit**), this edit is marked and given a unique identifier. That's how Git tracks edits: modified file(s) are selected (**git add**), and snapshots are created (**git commit**) with a unique identifier to track it.

GitHub is a website where all those edits can be hosted in the cloud, instead of just living on your computer. This makes it easier for other folks to edit them, too.

Let's say you're working on a group project, and that project has 10 different files. Those 10 files live in a **repository (or "repo")** on GitHub. A repo is just a folder containing all the files for your project. It also contains all the *history* of your project, too.

Normally when you start a project you'd create a new empty repo, with no history or files, but because this project is already in progress you don't want to start a new one. Instead you'll **fork** and **clone** the existing project repo, which means you copy *everything* in that GitHub repo —all the files and their history—onto your computer. This version that you now have is called your **local copy** or **working copy**.

So now, with this kind of version control in place, whenever you make edits to a file that's part of the repo that you just cloned, you can edit those files in your working copy. At the same time someone else from your team can also edit those same files, but on their working copy. No more waiting to send emails of files back and forth.

Let's say the project has been going for a while, and there's like, 230 revisions (**commits**) that have been made to the files over the course of its history. A revision is the state of the repository at a certain point in time. If someone breaks everything by uploading bad code, you can roll back (**git reset**) to a prior revision.

Okay, so you're editing files in that project. Git notices you've changed your working copy files, but it's only noticed that the files have changed, it hasn't done anything with that information yet. In order for Git to save a snapshot of all the changes, so that you can roll back to that save state or alert your colleagues that you've changed them, you have to **commit** those edits.

Once you've committed, you can then choose to share those files with the world, and get those edits/commits off of just your computer and put them onto GitHub. In that case, you'll **push** those commits (all the files you've tracked during edits) to GitHub, where you save your

changes back to the repository.

Usually this occurs without any problems.

Sometimes, however, you'll make changes to the same file that someone else also changed, committed, and pushed.

In the example case we're using here, you were working off revision #230, but I was editing it too, and I pushed my code before you did so the official GitHub repo for our project is actually now on revision #231. And the official Github revision #231 looks different from the edits you made to #230… so your revision #231 is different from the official repo #231.

So you now have to **pull** the new revision #231 from the GitHub repo, and you need to then **merge** your edits with the #231 version on GitHub. That is, you combine two sets of changes to the files in your project.

If you happened to edit the same exact lines of the same exact files as the version #231 I created, this creates a **merge conflict**, and someone has to step in and look at what those differences are and decide how to combine them together.

Once you do this, you can **push** this new revision which is now #232, because it combines the official #231 with your #231 to create a whole new version, which is a **child** of both #230 and #231. This idea of being a "child of" is part of what's called a **dependency graph**, where you can only push code that is a direct **descendant** of the official repo. Because you had version #230, but I changed #230 to become #231 on the official repo, your edits were not a direct descendant of the official repo number (now #231), so they had to be folded in.

If you want folks on your team to look at your edits, make comments on them, review your code, etc. you can initiate a **pull request** on GitHub, which tells people "hey, pull this new revision, test it out, and let me know what you think."

If you want to try out some side experimental thing—like adding some new feature—without breaking the official code base for everyone else, you can create a **branch** off of the repo at some point in its revision history. This lets you and your team mess around and try things out

without breaking the official version. Once you all are happy with things, you can then merge that branch back *onto* the main branch to fold in all of that branch's changes, history, etc. into the main repo.

Or, you might decide it was all terrible and no good, so you can change back to, or checkout, the main repo again, abandoning your changes.

Okay, that's basically all of Git and GitHub that you'll likely ever need to know. But the time you'd need to know more, you'll be expert enough that learning it will be much easier.

The lingo:

- **Git**
- **GitHub**
- **repository** (repo)
- **add**
- **clone**
- **working copy**
- **commit** (revision)
- **reset** (roll back)
- **push**
- **pull**
- **merge**
- **fetch**
- **merge conflict**
- **child**
- **dependency graph**
- **descendant**
- **pull request**
- **branch**
- **checkout**

Tags: March 2021

Categories: git

Updated: March 4, 2021