Tooling

Neil Kingdom

January 3, 2024

Contents

1 Preface

2 Introduction

I would argue that understanding the syntax and APIs of a language is only half the battle towards becoming a valuable developer. This is because programming is more than just understanding syntax and memorization. It also requires being able to leverage your own productivity. Developers are expected to learn many tools which can be beneficial for increasing productivity and which can aid in minimizing complexity. In this document, I will be focusing on three tools which I use on a regular basis and which I think are worthwhile additions to your arsenal. These three tools are git, make (as in Makefile), and GDB. If I could add a fourth, it would certainly be vim/neovim, however, vim is a very complex beast that is best learned through consistent usage. This list is, of course, heavily biased towards the work that I do, but even if you do not end up using any of these tools, it is still extremely beneficial to your growth to sit down and learn a new tool.

3 Git

I'll begin with arguably the most difficult, but also most widely used tool of the bunch. This would of course be Git. I'm sure you've heard of Git, but if not, I'll elaborate. Git is a version control system (VCS) written by Linus Torvalds (the same guy who made Linux) in 2008. Its predecessors include Subversion (SVN) and TFS. A VCS has multiple practical usages, but true to its name, it is predominantly used for tracking changes (or versions) over time. Git keeps a robust log of diffs known as the branch history. A diff is just a file that highlights changes since the previous version i.e. additions, removals, or modifications. One of the greatest features of Git is that it makes collaborative projects much easier since it will prevent two people from accidentally making conflicting changes. Instead, it forces both parties to resolve the conflicts before the repository is actually updated.

There is a big distinction to be made between Git (a command) and GitHub (a service hosted in the cloud which stores Git repositories remotely). Git is local to the user's machine, meaning that everything is tracked locally by default. GitHub is often the defacto choice for a remote Git client, however, others exist such as GitLab, BitBucket, SourceForge, and GNU's Savannah. I won't argue for which of these you ought to use, although I will say that latter two options are not mainstream choices.

Personally, I use GitHub almost exclusively to backup my personal projects and sometimes even other kinds of files. They offer a generous amount of storage space (I'm not even sure if there is a limit to be honest) completely for free!

3.1 Understanding Git

Git is actually quite difficult to learn, and even harder to master. It really does require a bit of theory and practice before you're able to get comfortable. Although many IDEs include visualization tools for Git, I highly recommend learning the command line tool instead.

First, I'll cover a bit of semantics. A repository is the logical collection of files that are to be tracked by Git. For all intents and purposes, this usually equates to your project's directory. Git does have support for submodules which are essentially just a way of nesting repositories within other repositories, but I'll cover that much later. Anyhow, all of the metadata used to track the repos history is stored within the .git directory. It is generally a good idea to not touch anything in here at the risk of severely messing up your project.

In order to create the .git directory i.e. setup the repo, we simply navigate to the root of the project or whatever it is you want to track and run git init. Assuming that files exist in the repo, we can try running git status. This subcommand will output some useful information. It will first state that we're on the main branch. One of the most interesting features of Git is its ability to switch between branches, which are effectively

diverging histories. Think of it like a multiverse of project histories that we can switch between. You can try implementing a change in one branch and then revert back to the previous branch to reset the project's internal state back to how it was on that branch. The default branch used to be called master but is now called main for political reasons. You can change this within Git's config files. Another piece of information output by Git is that there have not been any commits yet. A commit can be thought of as a checkpoint marking a significant set of changes or milestone within the change history. If you've ever dabbled with emulators, I like to think of commits as a save-state because we can jump between different commits. Assuming that there are files within your repo, you should also have a log indicating that there are one or more untracked files. Git does not keep track of any files by default, therefore, we have to opt in to which ones it should track. Git will propose that we use git add to track a file. In many cases, we just want to track all files, which can be done by running "git add .". The period is similar to the kleane/wildcard operator.

When we add files to be tracked by Git, they are queued into what is referred to as the staging area. The staging area simply contains files which are ready to be committed. Running git status again after we've added one or more files to be staged will yield a similar output, but now it should show all of the staged files under changes to be committed. It will also suggest running git rm –cached to remove a tracked/staged file. Never run git rm -f unless you want to actually delete the file from disk. If you remove a file with the regular rm command, it doesn't automatically make git forget about it, so you'll still have to run git rm –cached file.