Parth Patel

CSCI 3327

What is Git Workflow?

First and foremost, before delving into Git, one question must be answered: What is version control? Version control is one very popular method of managing and keeping track of your changes that you make to your source code. Essentially, a version control software (VCS) keeps track of these changes, which later allows the team and other users to check out the source code to understand the progress that has been made, or to revert back to older code when coming across errors. Whatever the reason may be, all developers would want to keep track of the history of their code in order to ensure that their end goal is accomplished. Generally, VCS organizes the source code in a “file tree” structure, which allows concurrent development, instead of only allowing developers to make changes one at a time, which would be very frustrating to deal with when working on a project, but the tree structure also enables teams to maximize their productivity.

These aren’t the only benefits of VCS. In fact, VCS allows for branching and merging, which is applicable regardless of whether you’re working in a team, or if you’re working alone. It is extremely useful to be able to have multiple streams of work happening at the same time, which you can then *merge* together once you decide that the time is right. Each stream of work is called a *branch*. Having branches also allows the user or the team to be able to keep track of the work that each specific user may have done, which is crucial when debugging. Another key benefit of VCS is that it has *traceability*, which means you can actually annotate each change that you make to your code. This lets the user and the team be able to backtrack and figure out *which* changes occurred and *where* they occurred, which is yet another important aspect to debugging. The amazing thing about traceability in a VCS is that it can hold *years* worth of progress and changes made to the source code. All of these benefits give the developers the ability to agree on a shared workflow that will work well for everybody, in order to maximize efficiency and productivity.

So, what is Git? Git is the most popular VCS today, and it was originally developed as an open source project in 2005 by the creator of Linux: Linus Torvalds. Majority of software projects today depend on Git to maintain productivity, efficiency, and organization. Git actually has a “distributed architecture”, so it can be classified as DVCS (Distributed Version Control System). Basically, instead of storing the entire version history of a project in one place, Git allows every developer to have a copy of the repository with the code, which contains its entire version history. One main strong point of Git is its performance. Git has implemented effective algorithms to store data as efficiently and as organized as possible, which will help save a lot of time for the developer. Git is also very secure and the repositories are actually secured with a hashing algorithm called SHA1, which helps keep out perpetrators, but also ensures that the code you have is in fact *your* code and that it has not been tampered with. One other key strong point of Git is its flexibility. Git is just as effective in both large-scale projects, small-scale projects, and can work with a variety of systems and protocols. Git enables the developers to branch and tag as well (which aren’t necessarily the best in other VCS’s, like SVN), and these changes are saved as a part of your project’s version history on Git. Aside from all of the benefits of Git, it is basically mandatory to have a Github if you wish to enter the realm of computer science, since just about every job or position will require you to be well-versed in Git.

One of the core features of Git is its repository. A *repository* (*repo* for short)is essentially a place to safely store your project, and it will also keep a history of the changes and versions of your code. A repository can be created through the *git init* command in the command prompt, which will initialize a repository. Once a repository has been created, it can be *cloned* to create a copy on your local computer. This is typically performed only once per repository, since all of the future commands and operations performed on repository will be managed through the cloned repository. After the repository has been cloned and initialized, you can *commit* changes to it. First, you would have to perform the *git add <insert file>* to add the file to the *staging area*. From there, you can do the *git commit -m <enter message>* command to commit the work to the repository and also attach a message to the commit, which can help later on when backtracking to previous versions. After this, you can upload your new changes from your local repository to your remote repository on GitHub by using the *git push* command.

The pull command for Git, which is *git pull*, is used to fetch the content from a remote repository and download it to your local repository, thus updating your local repository with the new content. In fact, the pull command is actually a combination of both *git fetch* and *git merge*. The fetch command is used to download the content, and then the merge command is used to merge this content into your local repository. The merge actually *commits* it to your local repository. Being able to use both push and pull commands are essential to the everyday Git workflow.

A branch in Git is fundamentally an independent stream of development. There is a command called *git branch* that allows you to create branches, and using these branches is crucial for the Git workflow, since it will be used with the *git checkout* and *git merge* commands frequently. The *git merge* command is what you use to put all of the forked history back together, which means taking the various branches that you’ve created and cohesively integrating them back into one branch. However, if you try to merge two branches that were both changed in the same location or same part of one particular file, then Git will run into a merge conflict, and Git will actually cancel the merge commit immediately before it happens, so that way the merge conflict can be solved manually. One last important point is: what is the Git workflow? The general flow of Gitflow actually goes step-by-step. First, you develop a branch from the main branch. Then, a release branch is created from the develop branch. After this, feature branches are created from the develop branch. Once the feature branch is complete, it is merged into the develop branch. Once the release branch is complete, it is merged into the develop branch and the main branch. If an issue occurs on the main branch, then a hotfix branch is created from the main branch. Finally, once the hotfix branch is complete, it then merges back into both the develop and main branches.

Works Cited

Atlassian. (n.d.). *Gitflow workflow: Atlassian Git Tutorial*. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow