Ahmed Malik

GitHub Essay

Git is known to be the most commonly used version control system in this day in age. Git has the adaptability, speed, and stability required to thrive in fast-paced markets. Git workflow is described as being a “recipe or recommendation” for how to use Git in order to complete all your tasks promptly. Git itself gives users a lot of individuality and allows them to manage changes however they’d like. There isn’t a single standardized process on how to interact with Git. In a team setting, it is imperative that you all agree on how your Git workflow should be developed.

There are quite a few choices you can pick from to make sure your team is all on the same page. The first would be the Centralized Workflow. This is recommended for teams transitioning from SVN. They are similar due to the use of a central repository to serve as the single point of entry for all changes during your project. The default development branch is called “main” instead of “trunk”. Switching from Subversion to Git isn't as complicated as it sounds, since your team can develop projects in the exact same way.

But of course, with anything there are going to be advantages and disadvantages. Choosing Git gives every developer their own local copy of the entire project. This allows each developer to work on their own on all the changes to the project. They can work at their own pace and do what is necessary to them at any given time. Git, in contrast to SVN, branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories. This Centralized Workflow is best when used for teams switching over from SVN to Git and also for smaller teams.

Looking closer, we have to break down how this all works in order to actually use it. Developers can start off by cloning the central repository. They edit files and commit changes similarly to that of SVN. But, with Git, these new commits are stored locally. This is great for developers because they can defer synchronizing upstream until they're at a good spot for a break point. In order to publish changes to the actual project itself, developers “push” their local “main” branch to the central repository. This is comparable to “svn commit”. The only difference here is that it adds all of the local commits that aren’t already in the central “main” branch.

Someone needs to create the central repository on a server to start everything up. For a new project, initializing an empty repository is key. If you don’t, you’ll have to import an existing Git or SVN repository. It is important to note that central repositories should always be bare repositories, which means they shouldn't have a working directory. After all this, each developer creates a local copy of the entire project. You can do this by using the “git clone” command. When cloning a repository locally, a developed then can make changes using the standard Git commit process that is set in place. This means: edit, stage, and commit.

When the local repository has new changes committed, the changes will need to be pushed in order to share it with other developers on the project. The “ git push origin main” command will push the newly committed changes to the central repository. Git will output a message if there is a conflict regarding updates from another developer that contains code that

conflicts with the intended push updates. If this does occur, “git pull” will be the first thing you do.

The central repository resembles the official project, so its commit history should be viewed as “sacred and immutable”. Git will simply refuse to push changes if a developer’s local commits diverge from the central repository. One of the best things about Git is that anyone can resolve their own merge conflicts. If any of the developers do run into an issue, they can run a “git status” to see exactly where the problem is located. Once they are satisfied with their results, the developer(s) can stage the file(s) and let “git rebase” do the rest of the work.

If your team is confident with the Centralized Workflow but really wants to hone in on collaboration efforts, they may want to check out the Feature Branch Workflow. Feature branching is simply an extension of Centralized Workflow. The main idea behind it is that all feature development should take place in a dedicated branch instead of the “main” branch. This just makes it a lot easier for multiple developers to work on a certain feature without disturbing the main codebase. It also means the “main” branch should never contain broken code, which is a big plus for continuous integration environments.

Another common workflow is Gitflow Workflow. It defines a strict branching model designed around the project release. It doesn’t add any new concepts or commands beyond what is required for the Feature Branch Workflow. Alternatively, it assigns super-specific roles to different branches and defines how and when they should interact.

A workflow that is different from all of those mentioned above would be the Forking Workflow. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. In short, this means that each contributor has two Git repositories consisting of private local and public server-side. Choosing the Git workflow that works best for your team is crucial for any project.