Git and Github Reference Material

Contents

[1.0 What is Git / GitHub? 3](#_Toc56282421)

[2.0 Structure of Git 3](#_Toc56282422)

[3.0 Common Terminology 3](#_Toc56282423)

[4.0 Git Setup 3](#_Toc56282424)

[5.0 Git Commands 3](#_Toc56282425)

[6.0 Git Workflows 3](#_Toc56282426)

# What is Git / GitHub?

Definition for Git from Wikipedia:

“Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.”

Essentially, Git is a program that tracks code changes and allows people to work on different parts of a code base. It can also be used to identify and discuss bugs or code changes before they enter production. For Team ASTRAS, this accomplishes a couple things:

* Our code is safe. We can revert changes and have a clear revision history.
* Multiple people can work on the same code base without conflicting issues.
* If needed, we have a center for coordinating tasks and fixes.

GitHub is a website that hosts remote Git repositories. This is essentially the ‘cloud’ in which our files are stored. When we want to do some work, we pull this remote repository to our local computer, make our changes, then push our updates back to the remote repository.

*Why do I need to learn all of this? Wouldn’t it be easier to just do on Teams?*

We certainly could do all of this on Teams – however, there are a couple reasons to use Git instead:

* Version control – to keep records of versions on Teams, we would either start adding folders or files with those names. Before long, we will have a ton of content that is very obscure in how it has changed. With Git, you add a comment line on every ‘commit’, or update, that keeps track of changes.
* Anybody can pull the latest repository from GitHub. This includes the raspberry Pi. Otherwise, we would need to constantly move files from our computers to the Pi or develop directly on the Pi. Both of these options, in my experience, aren’t great.
* It is an industry standard. It is something you can add to your resume and will look great. Programming is something that is becoming more prominent in every industry, and Git/GitHub is one of the largest hosts for collaborating with programmers.

Expect to take a few hours out of your day to learn how Git works – I’ve tried to make this document as straight forward as possible and teach you just enough to work with it.

# Repositories

GitHub is where our code is stored on the cloud – this is our **Remote Repository.** This is where we pull the current code from and push our changes to. The remote repository is also called **origin.** Our remote repository is found at <https://github.com/Team-ASTRAS/Prudentia>.

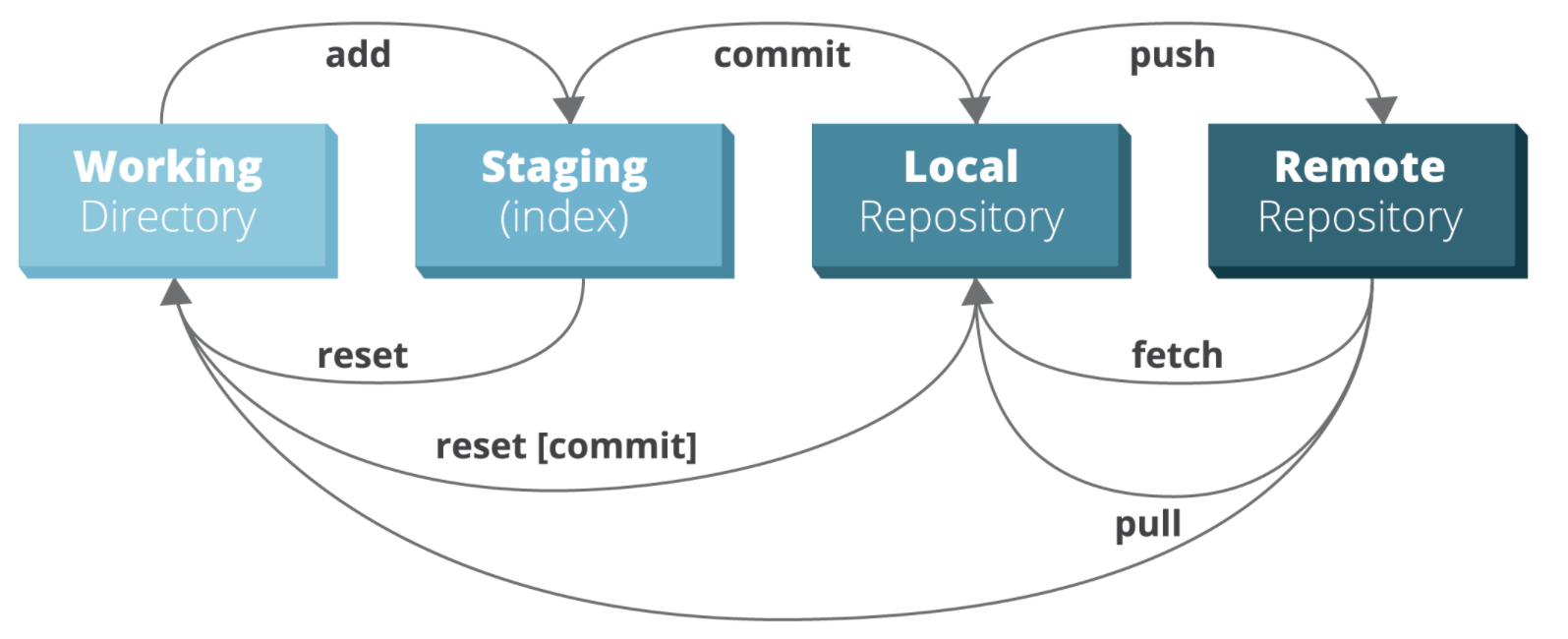
While you can make changes to the remote repository directly, typically you copy the remote repository to a **local repository.** This is a repository that sits on your computer, and changes to this repository will not affect the origin until you explicitly push those changes to origin.

Repositories keep track of all your files and changes. You cannot change your local repository’s current state directly; instead, you update files in the **working directory.** This is a file (which can be found under file explorer) that will have all your files and folders in it. From here, you can add/edit/delete. When you are ready to apply your changes (known as a **commit**), you first add your modified files to the **Staging Area**, then commit those changes.

Again, modifying the local repository does not affect the remote repository automatically. To make your changes visible on the remote repository, you **push** your local repository to the origin. This will update the remote repository with your latest committed local repository.

So, that’s a lot of text to take in. Below is a figure to show what this looks like graphically.

**GitHub Repository Structure**



Let’s walk through a scenario; a user logs on for the day and wants to edit something in the remote repository.

* First, the user **pulls** the origin to update the local repository with the latest files.
* Next, the user makes edits within the working directory.
* Next, the user **adds** those edits to the staging area.
  + If the user wanted to remove something from the area, **reset** can be used to revert this.
* Once all desired changes are added to the staging area, the user then **commits** the changes to the local repository.
  + Every commit is tracked – changes from any commit can be seen or reverted to.
  + If the user wanted to revert the repository to a previous commit, **reset <commitID>** can be used.
* Finally, the user could **push** the local repository to update the origin.

Examples of how this is done will be shown in later sections, for now the important thing is to know:

* How repositories are structured.
* Use pull to update your local repository from the origin.
* Use push to update the origin from your local repository.
* The process of applying changes in your working directory to a repository is committing**.**

# Branches and Merging

Branches are different versions of a set of files that can be modified separately from other branches, then later those changes can be **merged** back into the original branch. This functionality allows for different teams to work on different parts of the same project without interference. Many models exist – we will be using a model that has two main branches – *master* and *develop*.

The master branch is the area where production ready code exists. For us, this would be “release” versions of Prudentia’s software. Subsequent releases might add new features or fix bugs, but we only update the master when we are confident it works as expected.

The develop branch is the area where our work-in-progress (developing) code lives. This branch is committed to frequently as development continues. Once this branch reflects a production ready state, we **merge** the develop branch into the master branch to update it. The below graphic shows how the relationship between develop and master might look over time.

**Relationship Between Develop and Master Branches**



Shown above, each commit is a node. Develop commits are shown in yellow while master commits are shown in blue. Arrows that cross from one branch to another indicate a merge.

These two branches will exist indefinitely under this structure. Additional ”feature” branches can also be made off the develop branch if a large feature is being worked on and you don’t want interference from the develop branch.

For example, we may want to create a “feature” branch based on develop for the addition of the camera’s functionality. This allows us to do implementation and testing on our own branch without interfering with the develop branch. Below is a graphical representation of how this might look.

**A Feature Branch Based on Develop**



The feature branch shown in pink, and only exists as long as the feature is under development. Once it is merged back into develop, the feature branch is deleted.

This structure is largely based on this article: https://nvie.com/posts/a-successful-git-branching-model/

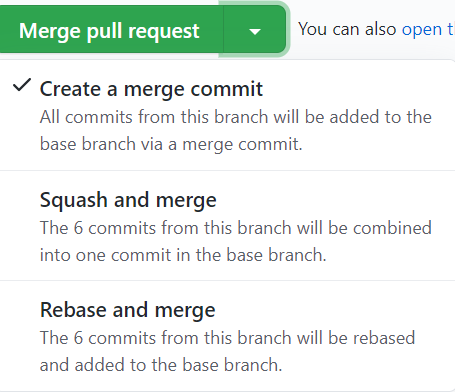
Feel free to investigate the article if you’d like some more background on the structure. Note that we are using a much more simplified system that only includes **master, develop, and feature branches.**

One additional function you should know of are **pull requests**. You can create pull requests when you are ready to merge a branch and this sends everybody involved a notification that you are ready to review your changes before the merge happens. This also lets everybody know that they should pull the updated branch after the merge is finalized.

Going back to the camera feature example, it might work like this:

* Create a new branch called *develop/camera* based on *develop*.
* Do some work in the *develop/camera* branch until it is implemented and tested.
* Create a **pull request** for *develop/camera* to be merged into *develop*
  + Here, all the commits from *develop/camera* are shown and you can leave additional comments on what you’ve done.
* Team members can then review the open pull request on GitHub. Once everything looks good, the pull request can be merged.

Note that there are several ways to merge a branch shown below. The differences in these merges are what sort of commit information is kept in the destination branch. For our purposes, you should only need to use “Create a merge commit” which will add every commit from your merging branch to the destination branch. If you’d like to learn more about different merge types, see <https://docs.github.com/en/free-pro-team@latest/github/administering-a-repository/about-merge-methods-on-github>



Examples of how branches are created, modified, and merged will be shown in a later section. The important takeaways are:

* Our architecture uses the *master* branch for production ready code, *develop* for work-in-progress code, and *feature* branches for large feature implementations.
* Pull requests are used to let the team know you are ready to merge your changes.

# Gitting Git Setup

(Excuse the horrible pun)

Git can be setup in a variety of ways. It can be solely used from GitHub with a browser; however, you are only accessing the remote repository in this way. To make full use of Git and our architecture, you have two options:

* Git bash
* GitHub Desktop

Git bash is entirely command line based. GitHub Desktop is a GUI application that offers most of the functionality of Git bash (note there are many other GUI applications for GitHub). If you want the full experience in learning Git, I’d recommend learning Git bash as it will teach you the underlying principles of Git. For what we need to do, GitHub Desktop is much easier to approach and will be covered in this section. If you are interested in using Git bash, see this crash course on youtube: <https://www.youtube.com/watch?v=SWYqp7iY_Tc&ab_channel=TraversyMedia>

## GitHub Desktop Installation

1. First, download GitHub Desktop installer here: <https://desktop.github.com/>

Make sure to select your appropriate operating system. All information in this document assumes a Windows OS. While there should be little deviation with other setups, please let me know if you run into problems on another OS.

1. Run the installer and launch GitHub Desktop.
2. Sign in with your GitHub account. Ensure this account is the same GitHub account that was added to Prudentia.