Introduction to GitHub

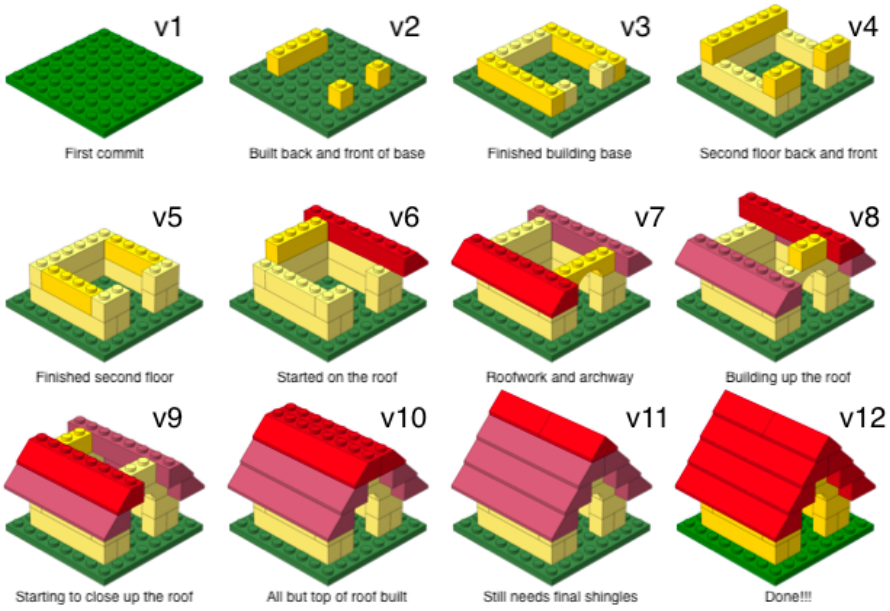
1 Introduction to Git

Git is an open source software for version control developed by the same guy that developed Linux, Linus Torvald. Git was developed to give software developers a protocol and system for version control that is fast and allows for collaborative work. GitHub is the most popular service built on the Git, its main value offering is that it hosts servers and streamlines the process of sharing and collaborating using the Git protocol, as well as having a GUI and desktop program that allows you to use Git without using command line. Git services can be also used separate of GitHub, either in competing services like GitLab, or in a self run Git server you can set up your self.

2 Git Features

2.1 Version Control

Version control allows you to create save states (called commits) that keep track of the changes made to your code base over time. These commits should be named according to the updates made to the code base, and give you the ability to return your code to any of these commited saved states for any reason (like breaking your code so it doesn't compile/run). Version control is not only useful as a personal tool, to act as changelog or to return to save states, but is also very useful when teaching the code base to a new user/developer. New users can see when certain features were added to the code, how they were added, and how the code developed.



2.2 Branches

While version control acts as saved states, branches are different universes where you can test changes or updates to the code base before either merging them together, or continuing a new project with different features on a new branch. After making a pull request, the current state of the branch you chose to pull will be reflected in your local working space, you can make as many commits as you want on this local working version of the branch before merging back to the branch. Problems only occur here if this branch has changed since you made the pull request, we will cover this issue later when discussing workflows.

Typically there will usually be a Main/Master branch, the latest fully updated and production ready version of the code, and development branches, which come off of the Master branch and merge back once all changes have been tested and approved. Additionally whoever is in charge of the project may want to have extra working branches, such as a test branch for potentially unstable changes or a bugfix branch dedicated to fixing bugs in Master. There is technically no limit to the number of branches you can branch off of a branch, and a significant number of popular open source programs exists as new working branches of abandoned Master branches.

2.3 gitignore

Git allows us to include .gitignore files in a repository, which tells git to ignore changes made to files in particular locations or with particular file endings whenever we make commits. For example, if sensitive information like filepaths or client information are stored in a file: `.\important_information\important_file.json` we can tell Git to ignore this file, and any changes made to it by either including the following lines in the gitignore file:

```
important_information\
```

to ignore everything in that folder or

*.json

to ignore all .json files in the repository.

2.4 Repositories

A repository is the folder that collects all commits and branches related to a single git project. You can import the latest changes made to the branch of your choice from the repository with a "pull" request or upload the branch you are working on with a "push". Additionally, repositories can also act a storage place for readme documents that explain the software as well as any non code files required in order to run the software. This can include almost anything as

long as it fits within the maximum storage limit for your project. I have not yet encountered a file type that could not be stored in a GitHub repository.

There is, however, one distinction that should be made when working in collaborative projects. Branches in the local repository and the server side remote repository (the one on GitHub) are only in sync (and must be in sync) at the time of a pull request or a push request. After a pull request the branch you pull is mirrored in the client side local repository after which you are free to make changes and commits in your local repository as much as you want; these new changes will not be reflected server side. At the moment you push your client side commits will be uploaded to the server side branch and these changes will be enshrined forever in the remote repository (unless you have merge conflicts).

3 Setting Up Git

There are a few main ways to interact with repositories on GitHub: GitHub Desktop, through the browser client, or through command line with CLI. CLI is the most flexible and quickest while the GUI in the desktop client is easier and mostly foolproof. The browser client is mostly limited to pull requests, and dealing with administrative account settings. Additionally, most IDEs have built in GitHub functionality with either GUI or command line inputs.

3.1 Windows

While most linux distros come with Git preinstalled, Windows does not. Installing GitHub desktop Git is installed along with it. The second option is to install GitHub CLI, which allows you to use the linux style Git commands from command line and also installs Git to your system. The third option is to directly download Git, since Git has been open source for almost two decades now there are a few places you can get it. The one recommended on GitHub is https://gitforwindows.org/. This works with local Git repositories or remotely hosted ones such as GitHub or GitLab.

4 Working With Git

Once Git is setup and working on your system you can start working with it in any project. Projects that do not require remote repositories (such as individual projects or proof of concepts) can be run using a local Git repository. With git installed one can simply

git init project1

to create a git repository in the working folder and get started. If a remote repository is needed, the project will have to be connected to the remote repository, either manually in command line or through a dedicated UI such as GitHub Desktop.

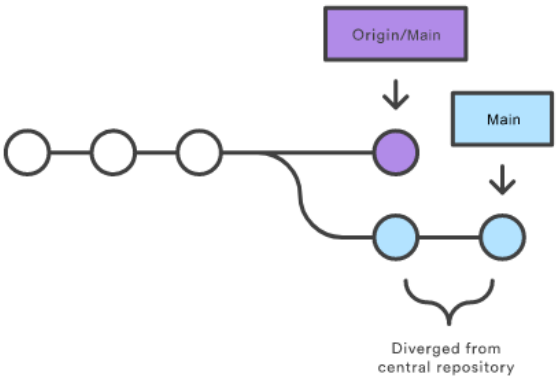
5 Example Git Workflows

While standardizing the Git workflow for a project is not necessary, it is very much recommended. A standardized workflow helps to clear up issues that might arise when attempting to merge changes from multiple different users to a single branch.

5.1 Everyone Works in Main

The simplest workflow, one often used in introductory or single developer projects, is working with a single main branch. In essence and in the best case scenario, this functions similarly to a google sheet, excel online, or overleaf document. Main acts as both the latest functioning version of the document and the development/working/production environment. However doing this in Git comes with a few major disadvantages, the largest of which is dealing with uncoordinated pushes from multiple contributors.

For example in a simple collaborating scenario with persons A and B, If persons A and B both pull from the main branch at the same time they are free to make changes and commit as much as they want in their local repositories. As mentioned previously git will not allow you to push changes to a branch if your local version diverges from the branch, for example if someone else pushes a commit after you've made a pull request, as shown below.



In this situation, person B will need to rebase by pulling person A's commits into their local repository, integrate these changes locally, then pushing to the remote. One way to do this is as follows:

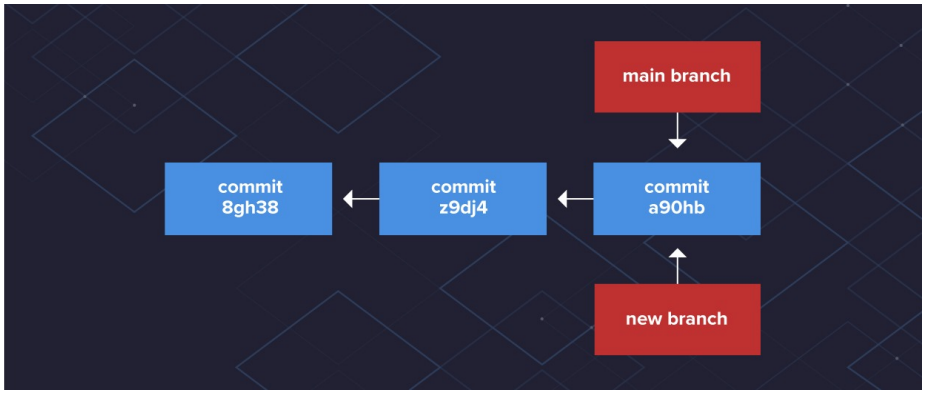
git pull – rebase origin main

Which will pull all of the latest commits in main into your branch, after which you can merge, commit, and push.

A second issue with having only one main branch is the lack of separation between development and production environments. One of the most powerful features of Git is the ability to keep a clean production environment separated from potentially messy production and test environments. Without this separation, it would become necessary to leverage the commit history in order to re-mediate issues such as code breaking changes, rather than having a clean master branch to compare changes to.

5.2 Every Feature Gets a Branch

One of the more common setups for medium sized teams, this is a setup that is slightly more complicated, but helps deal with issues that arise when multiple contributors are working on features at the same time, and the inherent issue of needing to rebase the repository if someone else pushes their commits after you pull.
Here a new branch is created whenever there is a ticket to create a new feature or functionality. The changes to implement the new feature are worked on in the feature branch, and are only merged with Main after they have been reviewed and approved.
The main issues that can arise from this setup mainly stem from merges/changes made to the main branch at a point after the feature branch was split. These new main updates will have to be reconciled with the branch by rebasing the feature branch to the newest 'main' before a merge can be completed.



6 SyntaxCheat Sheet

6.1 git init [repository name]

creates a new repository

6.2 git add [file name]

Stages changes before they are commited. Follow 'add' with the name of the specific changed/added files you want to stage, or just follow with a period to commit everything (This is generally what I would do).

for a specific file

git add changedfile.py

for all files

git add .

6.3 git commit -m [commit message]

commits the staged changes, don't forget to write a descriptive commit message

6.4 git pull

makes a pull request

6.5 git push

pushes current commits to the repository

6.6 git checkout [branch you want to switch to]

Switches branches. If you specify a commit on the current branch you're currently on (and a branch doesn't already exist with that name) then it creates a new branch at that commit point with the name of that commit.

git checkout -b [branch name] Creates a new branch called "branch name" that is a copy of the branch you're currently on.

6.7 git merge

merges branches together to, for example, merge a development branch to the main branch.