Part 1

Git Guide

What is Git?

"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" (Wikipedia)

Using Git Bash

For this guide, we would be using Git Bash – a CLI git toolkit. Using a CLI git toolkit makes it simpler for you to access git functionalities, as it is fairly lightweight and straightforward to use. It is also capable of using Linux shell commands which can make you familiar with Linux command-line interface that could really help you in industries as Linux is widely and sometimes strictly used there. Furthermore, debugging is quite efficient with a CLI git toolkit like Git Bash.

Basic Workflow

Note: This workflow can change from your flow of working in Git. Also, this is meant to be a very basic git guide targeted specifically for SMD assignments' requirements. In industry, people use different Git workflow **models** suited to their needs and preferences, so do check out those workflow models after reading this guide.

1. **Clone your remote repository**

   Since you had already created a remote repository (SMD Assignments) on GitHub website, you only needed to 'clone' (i.e. **downloading a copy of remote repository to your computer**) it. **Git init** wouldn't be used here as it is to convert a normal folder into a repository. As you already had a repository made for you online, you just had to bring it to your local system via command **"git clone [repository link]"**

2. **Initiate your project in the repository folder**

   Now, you have cloned a remote repository into your local system, it is advised that you start your new project directly into that repository instead of working on any other drive or folder and then copy/pasting files in the end. If you work elsewhere and eventually copy/paste your data to your repository, you would be missing out a lot of features that your IDE could provide you while working directly inside git repository folder. E.g. if any 'merge conflict' occur during sending your code to the remote repository, you would have to manually search for the files and line numbers in files where conflicts occurred.

   Hence, initiate/create your project inside the repository folder and work directly from there using any Git supported IDE such as IntelliJ

3. **Create new branch(es)**

   By default, you would be at master branch. Speaking of branches, Git is essentially a version control and collaborative system. To provide you these facilities, it uses **branches.** You can think of branches as separate flows of work independent to each other. Meaning, if multiple people work on same project and they want to work without any interference from other developers, they create separate branches so that their work gets separated from each other. Each time you update your branch, git tracks its history, so you can see that at what date/time you added/changed what files in your branch. If multiple people are working on a single branch, then, of course, the history would be mixed with everyone's updates, making it difficult for you to track only your changes. Hence, you create different branches and then in your own branches, you do your work.

   More on branches here.

   Now, about master branch, it is the one that has the final version of the code that is able to be used commercially, so code is only moved to this branch when it is completely developed and test. For development, we use separate branch(es), traditionally named **"develop".**

   You can create a new branch by using **"git branch [branch name]".**

   If you want to move into that branch (just like you want to move into a new folder when you create it), use **"git checkout [branch name]".**

   Tip: To create and move into the newly created branch at the same time, use: **git checkout -b [branch name]**

   For this assignment, you would:

   - Create a develop branch: "git branch develop"
   - Then send your local branch to the remote repository because at the moment, your remote repository would have no information about the new branch. You will do this by using: "git push -u origin develop"

     (Search about "-u" and "origin" online)

- Similarly, create two other branches (member1 and member2) and push them to remote as well.

- Finally, move into your branch to work there: “git checkout member1”

4. Commit your files

Once you have started working, there would be new files created or previous files edited or deleted and as such. These changes won’t be saved in your local repository unless you “commit” these. Basically, the repository folder you are working inside is not a normal folder but instead it is a git repository and it has a state. Initially, that repository would be in the state at which you cloned it from the website. To update its state i.e. to tell this repository that now it should save the added/edited/deleted files and change its state, you use commit feature of git.

Before commit, you need to add/stage your files that you want to commit – basically you tell git that which files it should save in the new state of repository. To add all the files in one go, use:
“git add *”

Then you locally update your repository by using: **git commit -m “any meaningful message”**

Now, your local changes and locally tracked and locally saved.

5. Push your changes to remote repository

The reason I emphasized the word “local” is because, at this stage, your changes are stored at local level. The repository you have online wouldn’t have any update unless you push (i.e. send local changes to remote) your work. To push your latest commits to your remote repository, use: **git push**

Initially, you would get any an error after “git push” like “fatal: The current branch has no upstream branch”. Simply use the suggested command and you will be fine. (Do a search online that why we got this error)

Once, your code is pushed to the branch you were working on, you can check your files on remote repository (on GitHub website) in the respective branch (member1). Other branches wouldn’t have these files. As you only pushed to the specific branch you were working on.

6. Merge your code

Now, your project is divided into two branches i.e. some files are present in member1 branch and some files are in member2 branch. Of course, to make your project work, you need to combine the data of both files. To combine the data of different branches, we use merge functionality.

- To merge the data in a branch, we first have to move into that branch where we want to bring our data. Here, we would use develop branch to have the merged data of both branches. So, use: **git checkout develop**

- Now to bring the data from other branch to develop branch, use: **git merge member1** and **git merge member2**

While merging, if working on same files, you can face **conflicts.** Conflicts occur when multiple developers make different changings to the same file. When it happens git couldn't know that what changes from which developer to actually save in the latest update and what changes to discard. Therefore, you have to manually resolve your conflicts in the files and then again add, commit, push and merge.

If using an IDE while your code is inside repository folder as discussed above, the conflicts would automatically be highlighted by the IDE.

- If there are no conflicts, then it means the data of both the branches (member1 and member2) is now successfully merged into the develop branch. But do note that these are local changes. Your remote repository has no information about what you have just done. In fact, neither your local repository has made an update. So to locally and remotely update your changings, again perform add, commit, push commands while being on the develop branch (because now you want to update your develop branch, of course).

Getting latest remote changes to local system

To get the latest changes of the remote repository, you use **git pull** command. Note that if someone has pushed to your branch, then your branch at remote level will be updated. Now at your side, when you will push your data to your branch, it won't let you do so. Because pushes/updates need to be ordered. As such, the push from other developer in your branch has changed your branch's state to let's say, state 1. At your side, you have also made changes, making your local state change to **state 1**. Now git won't allow you to push your code unless your first bring the remotely updated state of your branch (state 1 at remote). Once you use "git pull", the changed/deleted/files will be retrieved and your local branch would then be at the same level as of remote's (state 1). Now when you will push your code, it will allow it. The updated state of your branch at remote level after your push will be **state 2**.

As such:

System 1: Push (Branch A - local) -> State 1 (Branch A - remote)

System 2: Push (Branch A - local) -> State 1 (Branch A - remote) **(Won't be allowed)**

System 2: Pull (Branch A - local) -> State 1 (Branch A - local)

(Your local branch is now at the same state as the remote one i.e. System 1's changes would be retrieved in your system)

System 2: Push (Branch A - local) -> State 2 (Branch A - remote)

(Your changes will now update the remote branch's state to State 2)