Git & Github - Commandline & Collaboration

Git and GitHub are wonderful tools that can be a great help for any coder if used properly. Git can be installed on our local machine to track the versions of the code we write and GitHub can be used as a cloud platform to store them. As GitHub is a cloud platform used to store the written code, it can be used to collaborate with others while working together as a team for a single project. In this write-up, I will try to explain the basic Git operations using command line and how you can use Git and GitHub for collaboration.

If you are already familiar with using Git command line, you can skip reading *Git using command line* and continue with *GitHub - Cloud & Collaboration*

Git using command line

To make the best out of git, we need to have a better idea on how to handle git using command line.

Following are the most important & frequent operations and commands that we come across while using git.

Adding changes to staging:

The staging area can be considered as the preview of your next *commit*. When you do the next commit, all these changes added to the staging area will permanently be stitched to the **branch** you are working on right now (the concept of branch will be mentioned after a while).

Once committed, these changes cannot be removed from the *commit history,* although, you can revert to any one of the past commits.

The below command can be used to add the changes that you made in your local files to the *staging area*.

```
git add <file>
```

Eg: `git add .` *(To add all changes made after the previous commit to staging)*

These are the possible variations of the *add* command:

1. `git add .` : Add all changes you made after the previous commit to the staging area (The dot (.) indicates all files).

2. `git add foldername/filename1.js foldername/filename2` .css : Add the changes made to files indicate by the specific filenames. foldername/filename1.js and foldername/filename2.css are the paths to the two files which were changed.

Commiting the changes from staging area:

```
1 git commit
```

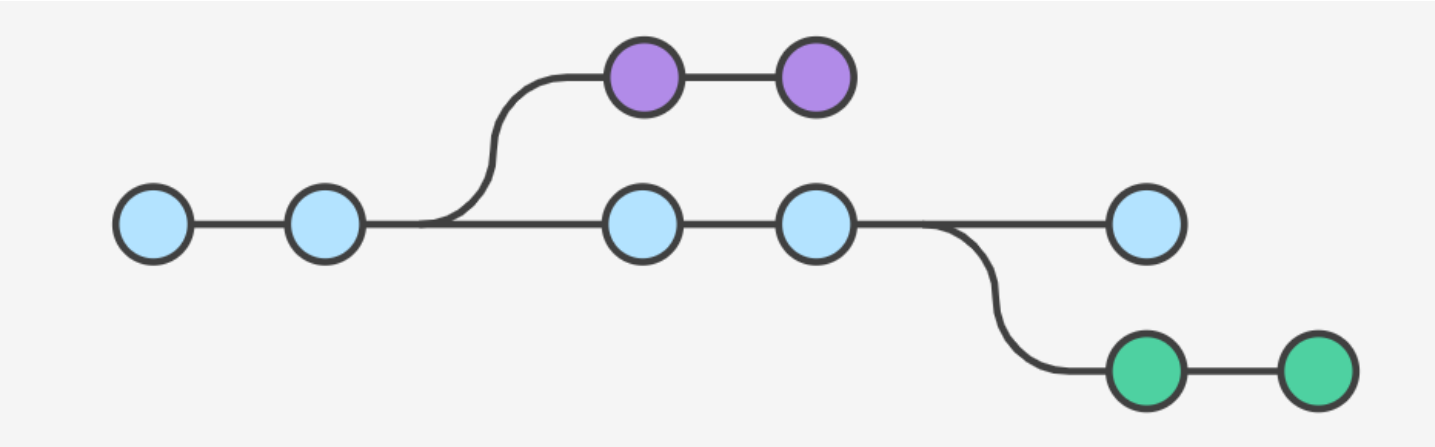
Eg: `git commit -m "commit message"` *(Commit the changes from staging area to the current branch.)*

The command can be used to commit the changes that are now in the staging area to the branch you are working on. If you are at branch *main,* the commits you do now will be stitched only to the *main* branch.

Here is a variation of the commit command: `git commit -m "commit message"`

It's better to use the commit command in this format, as -m is the message flag indicating that, the following is the commit message and running this command will add the commit message directly to the commit. If you use the command `git commit` without the message flag to commit, you will have to enter the commit message using command line editor, which is not user friendly.

Concept of branches:



Branches are one of the most important features of Git. Branches are used to work on your code without adversely affecting the code you've already written and committed. For simplicity, you can assume the creation of a branch, like creating a copy of your master project folder on your local disk. When you create a copy of the master folder, you effectively have two different copies of the same code. When you start working in one of the folders, the other stays the same and if you want, you can make changes on the other copy too.

Compare the creation of this copy creation with the creation of branches.

Assume that you have the branch *main* now and all the changes are committed to it . On creating a new branch you will have all the content from the *main* branch in the new branch too.

To create a new branch, use command:

```
git branch <new-branch-name>
```

To view all branches in your local repo, use command:

```
git branch
```

So, if you decide to create a branch named *new-branch* , you can use the command*:* `git branch new-branch` . And if you use the command `git branch` , you can see that you have two branches and the branch you are standing on right now will be highlighted.

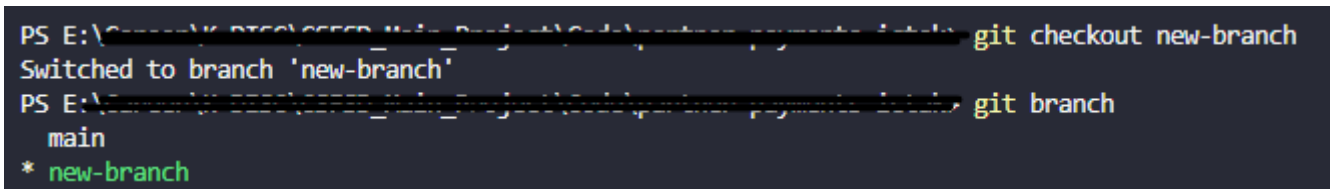
Now, you have two branches - *main* & *new-branch* - with the same content.

If you want to make sure that the changes you are going to make shouldn't affect the code you already have, you can **checkout** to the newly created branch and work on it. This is like working on the fresh copy you made by copying the working folder. The main files are going to stay the same.

To checkout to the newly created branch, use command:

```
git checkout <branch-name>
```

So, if you want to work on the branch *new-branch* and work on it*,* you can checkout it by using the command: `git checkout new-branch` . You can use `git branch` again to make sure if you are on the branch you need to be.



Now that you are on the *new-branch* you made, you can safely make changes to the working folder without affecting the *main* branch. Any commits that you do now will be attached to the *new-branch* only. If you checkout back to the *main* branch, you can observe that there is no change to the code and all the changes will be visible if you checkout to *new-branch.*

Merging Branches:

If you have made your changes and you are satisfied with it, you might want to include the changes you made to the *main branch.* After all, the *new-branch* was a temporary one you made to make sure that nothing messes up with the original code.

To merge the two branches, the first thing you need to do is to checkout to the branch to which you need to merge the changes to. In our case, we need to include the changes from *new-branch* to *main.* For this, use `git checkout main` to checkout to *main* branch. Now, you can use the following **command to merge changes from mentioned branch to the branch you are at now:**

```
git merge <branch-to-merge>
```

As we have already checked out to *main,* you can use `git merge new-branch` to incorporate all changes from the *new-branch* to *main* branch.

If some changes have been made to the *main* branch after the creation of *new-branch,* there are chances for *merge conflicts* if changes have been made to same files from different branches. This problem will be addressed later when we talk about collaboration, as when we work alone, it's rare that we make changes to two different branches together.

GitHub - Cloud & Collaboration

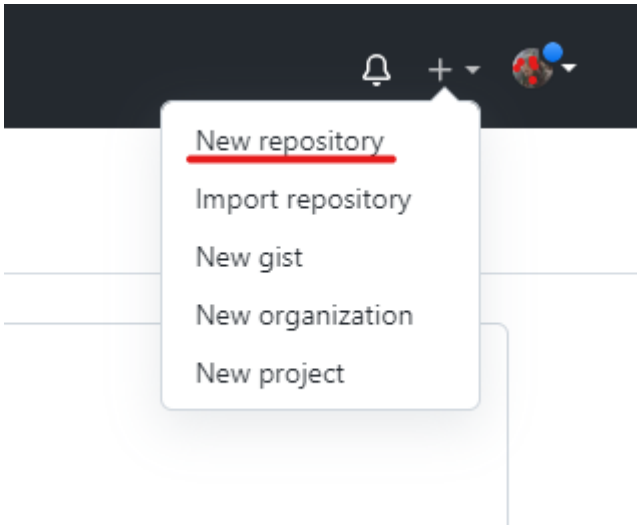
Until now, we have only looked at working with git in the local folder. GitHub is the cloud storage platform for storing git repositories so that we can have a backup of the local working folder in the cloud and also to collaborate with team mates if we are working as a team.

Pushing Code to GitHub:

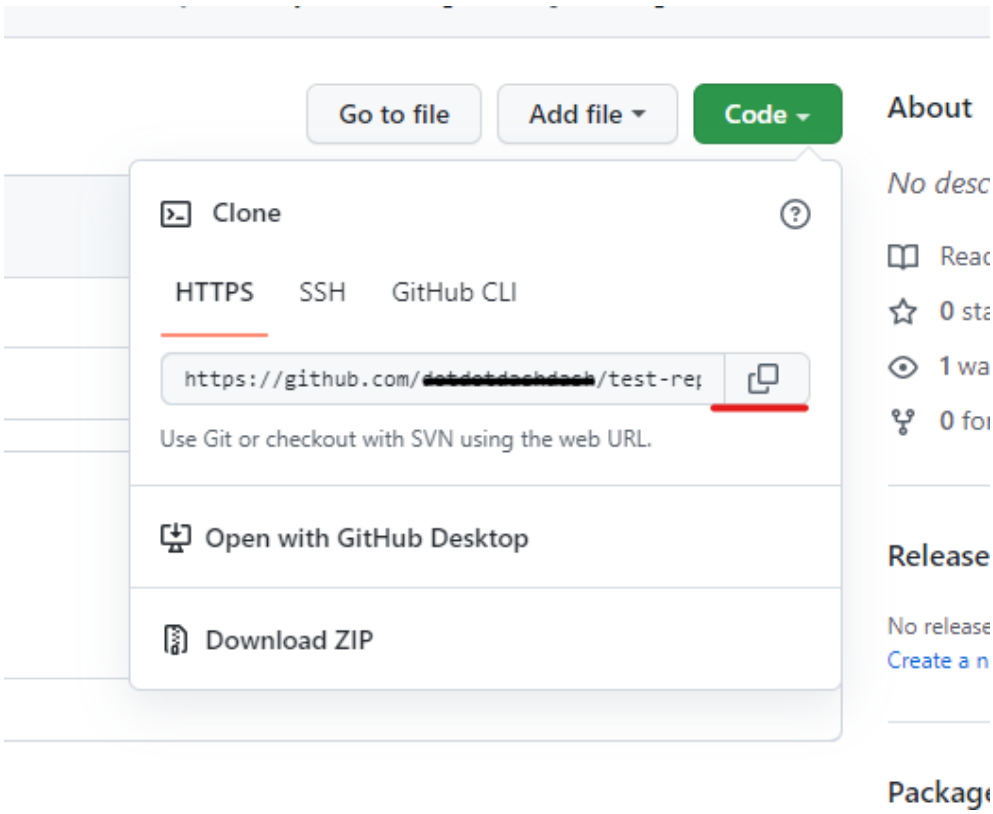
Adding a remote repository

To push to the local repo to GitHub, first we need to configure the GitHub repo to which we need to sync our local files with.

To do this, go to GitHub, sign in to your GitHub account, which you can create for free, and add a new repository:



Now open the created repo and copy the link to the repository after pressing the green colored *code* button, as shown.



After copying the remote repo link, go to your working folder, which is already a local git repository and then use the following command to configure the remote repo using the copied link and some name we choose for the remote..

```
git remote add <remote-name> <copied-github-repo-link>
```

Conventionally, *origin* is the name given for remote repo. So you can use the command: `git remote add origin <copied-github-repo-link>` to add remote repo with the name *origin.*

The word *origin* is nothing but a name given as a reference to the remote repo you configured right now. You can call it anything you wish, but *origin* is used conventionally.

Pushing the code to remote repository

Once you have configured the remote repo, you can *push* the local changes to the remote repo you just configured. You can use the following **command to push to the remote:**

```
git push <remote-name> <branch-name>
```

If *origin* is the name you gave to the remote repository and *main* is the branch you want to push to, use `git push origin main` to push to *origin/main*

If you haven't already signed in to github in local machine, it may ask you to sign in. Make sure you sign in to the same account you used to create the GitHub repository, or else you won't be authorized to push to the repo.

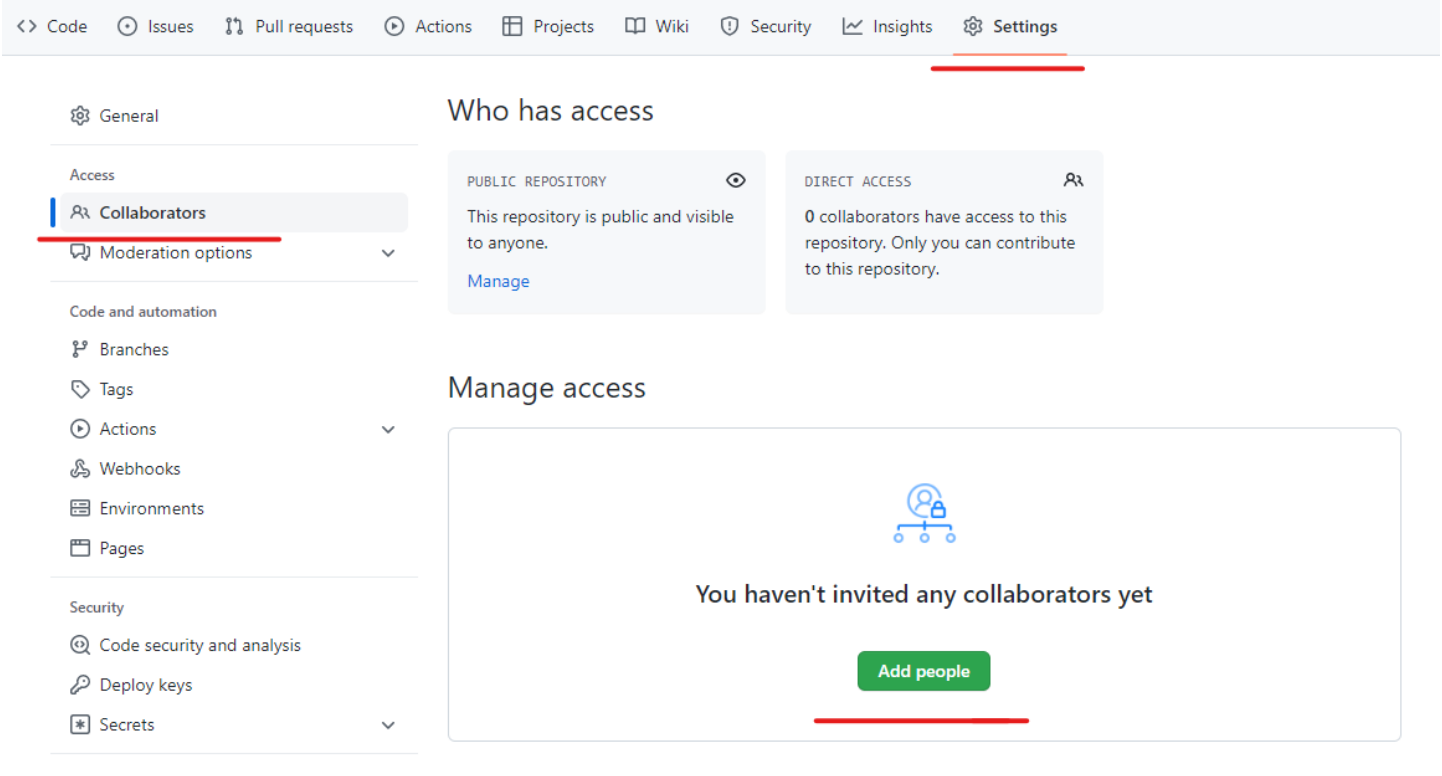
When you push the code to github, all the commits in your local branch you are at right now, will be pushed to the mentioned branch in the remote repository

Collaboration Using Git & GitHub:

When we work as a team, Git and GitHub are great tools to collaborate during the coding process. To collaborate, we don't need to know anything new other than those things discussed above. What we need to understand is the work flow to work as a team using Git and GitHub.

There are few things to take care of before we begin to work together.

First step is to create a central GitHub repository. This can be done by anyone in the team. Once the central GitHub repo is created, add the team members by visiting the settings tab in the GitHub repository.



Click on *Add People* button to add people using email or GitHub username. Once they are added as a collaborator they can push to this repo.

After adding the team members as collaborators, c*lone* this GitHub repo to their local machine by using the command:

```
git clone <central-github-repo-url>
```

The *clone* instruction will copy all content from the central remote repo to the local machine and set up the remote repo with the default name - *origin.*

After the team members cloned the central repo, you can start collaborative coding. Following is the general workflow after setting up for collaboration. All commands mentioned below has bee

Work-flow for collaboration

1. Pull the code from remote branch to make sure everything is synced. Use command: `git pull origin main`
2. Create a new local branch for every team member. Use command: `git branch new-branch`
3. Work in the new branch.
4. Pull the main branch again to make sure changes after creating the new branch are also synced (maybe someone else would have pushed to the main branch). Use command: `git pull origin main`

5. Standing on the *new-branch*, merge everything from *main* to the *new-branch*. Use command: `git merge main` . We do this by standing on the *new-branch,* instead of checking out to the *main* and then merge, because if you and someone else has changed the same file and he pushed it to the main, there are chances of conflict. If these conflicts occur in the *main* branch, it is a problem, as the main branch we work on could potentially be damaged by conflicts. So we stand on our personal branch and then merge the *main* branch to the personal branch, *new-branch.* So, any conflict, if any, would occur only on the branch we create for ourselves.
6. Resolve the conflicts, if any, by opening the files which are marked as conflict. After resolving them, commit the changes you made to your branch, *new-branch*. You may have to add the change you made for conflict resolution by using `git add .` & then commit them using `git commit -m "commit message"` .
7. Checkout to the *main* branch using `git checkout main` .
8. Merge everything from the *new-branch* to the *main* branch. Now no conflict will arise, as you have already resolved any conflicts that would happen now, in step 5 & 6, by merging *main* branch with your *new-branch* before checking out to the *main* branch. This way, we can ensure the integrity of main branch
9. Now you can push everything to GitHub remote repository by making use of the command `git push origin main` . All the changes you made are now synced to the *main* branch in the remote repo.

If you need to further secure the main branch (This is the case if the main branch is the production branch and any error would bring the hosted application down), you can protect the main branch by authorizing only one team member to push to the main branch. In this case, no one else other than the assigned, pushes to the main branch.

If that is the case, after step 6, the flow changes as follows.

After step 6:

7. Push the *new-branch* to remote repo using command `git push origin new-branch` . This pushes the branch named *new-branch* to origin and anyone can pull it from the remote.
8. Inform the team member who is authorized for pushing to *main*, to pull the branch you just pushed to remote. He can do it using the command `git pull origin new-branch` .
9. Now he can verify if your merge will cause any problem and if he is also sure, he can merge the new branch to the main by checking out to the *main* and then running the command `git merge new-branch`
10. He can push it to the remote using `git push origin main`

Alternatively, you can just push the changes to the remote repository in GitHub using `git push origin new-branch` and the authorized person can pull that branch using `git`

`pull origin new-branch` and he can merge *main* to the *new-branch* from his local machine by first checking out to *new-branch* using `git checkout new-branch` and then merge *main* to *new-branch* using `git merge main`. This merges main with the new changes to the *new-branch*. Then conflicts should be resolved and merge everything back to the *main* branch using `git checkout main` & `git merge new-branch` and then push it to GitHub using `git push origin main`.