GIT Assignment

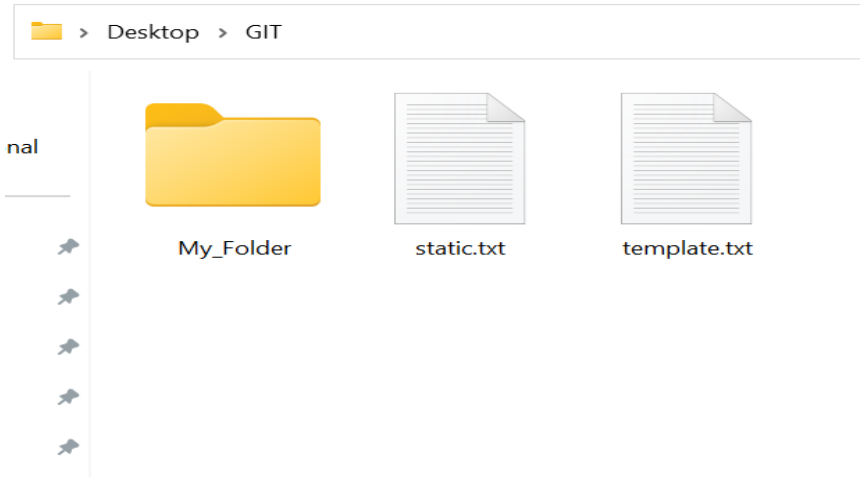
Distributed version control system (DVCS)
The Distributed version control system that we are using here is GIT.

What is GIT?

Git is currently the most popular version control system used by developers worldwide. It's a free and open-source tool that's very versatile and can be used for both small and large projects. You can use Git to track changes, collaborate with others, and manage different branches of your codebase.

1. Create a Distributed version control system, and upload files into it.

Created a folder GIT, having some files and directories



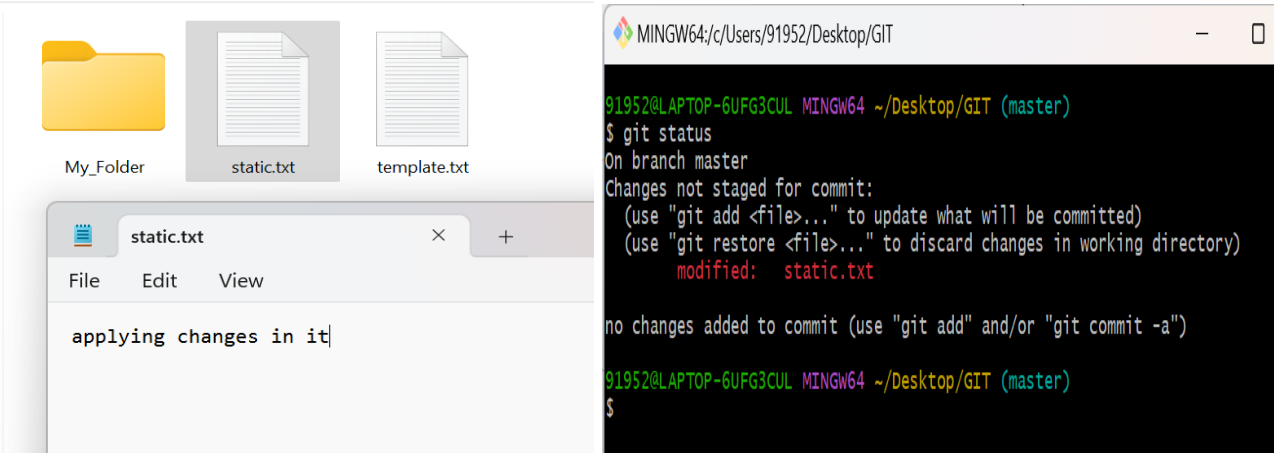
Upload the files in it.

1. `git init` *to initialize an empty Git repository.*
2. `git add .` *to stage all files in the directory for commit.*
3. `git status` *to check the status*
4. `git commit -m "first commit"` *to commit the changes with a message.*

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git init
Reinitialized existing Git repository in C:/Users/91952/Desktop/GIT/.git/

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git add --a

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   static.txt
        new file:   template.txt


91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git commit -m "First Commit"
[master (root-commit) 7b0c3bc] First Commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 static.txt
 create mode 100644 template.txt
```

2. Replicate a DVCS on your local machine, make changes to its files, and then put them back into it.

Applying changes in static.txt and checking the status



Add the files through git add - -a , then commit the changes.

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git add --a

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   static.txt


91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git commit -m "second commit"
[master 61f97f3] second commit
 1 file changed, 1 insertion(+)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git status
On branch master
nothing to commit, working tree clean

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$
```

3. List down all the files which have changed in the last commit. List only the file names.

Listing all the files

Use git log --oneline -2 to get the id of commits
Use git diff –name-only <id-2> <id-1>
Will give only the name of changed files in last commit files

4. You've accidentally added some files on a DVCS. Remove those files from the DVCS without deleting them from your local system.

Lets add files in it -(files1.txt, files2.txt) and check the status.

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git diff --name-only 61f97f3 7b0c3bc
static.txt

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fiels1.txt
        files2.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Create a file named .gitignore and put the names of files that we want to ignore (in these case file1,file2) And after that we will find that only .gitignore will be there to commit ,

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ touch .gitignore

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git add --a

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git commit -m "third commit"
[master d3f5b46] third commit
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

5. You've accidentally committed some files on a DVCS. Revert the DVCS to a previous stable state.

To revert to the previous commit, run the `git revert` command along with the commit ID of the current commit.

In our case, we'll be using the ID of the fourth commit:

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git log --oneline
9e20cfc (HEAD -> master) fourth commit
d3f5b46 third commit
61f97f3 second commit
7b0c3bc First Commit

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git revert 9e20cfc
[master b2a3fe1] Revert "comming back to third"
 1 file changed, 1 insertion(+), 2 deletions(-)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git log --oneline
b2a3fe1 (HEAD -> master) Revert "comming back to third"
9e20cfc fourth commit
d3f5b46 third commit
61f97f3 second commit
7b0c3bc First Commit
```

The command above will undo the current commit and revert the file to the state of the previous commit.

6. Create a DVCS capable of supporting 2 similar dev environments. Make changes in the files which are present on both of them and then merge the environments into one.

Create two branches for the two environments using git branch <branch-name>.
For example, git branch dev-env-1 and git branch dev-env-2

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch dev-env-1

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch dev-env-2

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$
```

Checkout the first branch using git checkout <branch-name>.

For example, git checkout dev-env-1.

Make changes to the files that are present in the environments.

Stage and commit the changes using git add <file> and git commit -m "<commit-message>"

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$ git checkout dev-env-1
Already on 'dev-env-1'
M       static.txt

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$ git add --a

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$ git commit -m "commit in dev-env-1"
[dev-env-1 f0663b1] commit in dev-env-1
 1 file changed, 1 insertion(+), 1 deletion(-)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$
```

Repeating same for dev-env-2

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$ git checkout dev-env-2
Switched to branch 'dev-env-2'

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-2)
$ git add --a

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-2)
$ git commit -m "commit in dev-env-2"
[dev-env-2 d1c865e] commit in dev-env-2
 1 file changed, 1 insertion(+)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-2)
$
```

Merging dev-env-2 into dev-env-1

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-2)
$ git checkout dev-env-1
Switched to branch 'dev-env-1'

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$ git merge dev-env-2
Merge made by the 'ort' strategy.
 template.txt | 1 +
 1 file changed, 1 insertion(+)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (dev-env-1)
$
```

7. You have a DVCS supporting your prod, test and dev environments, you need to make sure that only a particular set of commits from the test environment are added in the prod environment. What & how would you do it?

Create the prod, test and dev environments

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch pod

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch test

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch dev

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git branch
  dev
* master
  pod
  test
```

Now change some files in test and commit them

Then use log to get all the commits done

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (test)
$ git log --oneline
c3cb355 (HEAD -> test) third commit in test
7baa2c1 second commit in test
9aa6a59 first commit in test
```

Suppose we want second commit to be applied in prod environment

So use git cherry command to do so and using its hash id we got.

```
91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (master)
$ git checkout prod
Switched to branch 'prod'

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (prod)
$ git cherry-pick 7baa2c1
[prod 04c506c] second commit in test
 Date: Thu Mar 30 20:13:25 2023 +0530
 1 file changed, 1 insertion(+)

91952@LAPTOP-6UFG3CUL MINGW64 ~/Desktop/GIT (prod)
$ git log --oneline
04c506c (HEAD -> prod) second commit in test
b2a3fe1 (pod, master, dev) Revert "comming back to third"
9e20cfc fourth commit
d3f5b46 third commit
61f97f3 second commit
7b0c3bc First Commit
```

Scenario Questions:

1. **If the central repository is two commits ahead of your local repository, how will you push your code ?**

   Before pushing your code, you should first pull the latest changes from the central repository to your local repository. To do this, you can use the git pull command in your terminal:

   This will fetch the latest changes from the central repository and merge them into your local repository. If there are any conflicts between the changes in your local repository and the changes in the central repository, Git will prompt you to resolve them. Once you have pulled the latest changes and resolved any conflicts, you can then push your code to the central repository using the git push command:

   This will push your changes to the central repository and update it with your latest commits.

2. **I have two branches, A and B, where A is my main branch, and B is a the branch with new feature, describe the step by step process getting all of B's code in A**

   First checkout in B and apply appropriate changes

   Then in order to apply them, first stage the files and then commit them

   Then move back to main branch i.e. A (also called as master branch many times)

   Now use git merge command and all the commit or changes will be reflect in A now.

3. **Say you have to do an experimental change but still want to have a clean copy of your old code, should you use branching ? Explain your decision**

Yes, branching would be a good approach to keep your experimental change separate from your old code and have a clean copy of your code.

Here's why:

When you create a branch in Git, it creates a separate copy of your code that you can work on independently. This means that any changes you make in the new branch will not affect the old code in the master/main branch.

Additionally, branching allows you to collaborate with others on a specific feature or change, without affecting the main branch of your project. This makes it easier to work on multiple features or changes at the same time, without interfering with each other's work. Therefore, using branching is a good decision when you want to experiment with new changes while still keeping a clean copy of your old code