Git

It is a version control system. which saves all changes in a file and keep tracks of versions. because going into git command, let us first understand some basic terms.

Repository:

A git repository can be considered as a kind of database which contains all changes(versions) in a file and *metedata*. This repository is a single directory named as ".git" hidden in root directiory of project.

Clone:

Cloning means creating a copy of a repository in local system by downloading it from a remote repository.

Staging change:

Once changes are made one would need to *stage changes* before commiting. By staging, it means your are making changes to be ready to be committed. There might be case when, for example, 9 files have been changed. But a user need to commit only changes in 4 files. In this case user needs to *stage* only these 4 files.

Commit:

When all these changes are made and user considers it final, he may *commit* these changes. As soon as a commit is made a new version of our wokring project is created. A commit creates a *snapshot* of project version. should a user needs to restore a previous version, a commit know what changes and files are to be restored.
each commit needs to have a small message to briefly explain the changes.

Root or Working Directory:

This directory contains all files for the project in local computer. A VCS systems can be asked to populate a working copy of project of any version. However only one copy, not all version, of working project in local computer with a specific version.

Local repository:

This repository is located in local computer memory. Only a single user can use and make commits to this repository.

Remote Repository:

A remote repository is located on internet or other local server. A remote repository doesn't have a working directory but an exclusive ".git" repository directory. Remote repositories are useful when working in teams as everyone can access and pulich their changes.

untracked file:

the files which are not monitored by VCS for changes.

Git Configuration:

After isntallation of git some configuration is required. These configuration are user details. for this, open git bash and enter following command:

```
$ git config --global user.name “<username>”
$ git config --global user.email “<email>”
```

Now since we have installed and configured our git, we'll start the basic workflow of git VCS. Git VCS workflow have following steps:

1. initialize an empty project or clone exisiting repository.
2. Make Changes; add, edit, change files.
3. Stage changes.
4. commit changes

Summary of git:

Most commonly we'll need only a few git commands, which are, given in order:

1a: git init :

initialize and empty repository. to be used only once

1b: git clone <remote repo address>

clones a remote repo. to be used only once

2: git add <filename> :

stages changes.

3: git commit -m "commit message" :

commits the staged channges.

for remote repository:

1: git remote add origin <repo address> :

adds remote repo with *origin* aliasis. to be used only once.

2: git pull origin master

gets data from remote repository

3: git push origin master

send commited changes to remote repository

1a. Initialize an empty Repository:

In order to do this, open git command line(git bash) and go to required directory using cd command. for example, I wanna create a repository in `D:\PIAIC\gitPractice\Project1` , which will be root directory. we need to run following command:

```
$ cd /d/piaic/gitpractice/project1
```

as one may notice that path name is not case sensitive which means *gitPractice* and *gitpractice* are same.
Once in required directory, run the follwoing command to initialize repository:

```
$ git init
```

It will initialize an empty repository and a hidden folder named with ".git" will be created in the root directory. The contents of a directory can be listed by using following command in command line:

```
ls -la
```

1b. Cloning an existing repository.

If there is an existing repository, let's say a remote repository, one may create a local copy using clone command. it can be done by using following command:

```
$ git clone <remote Repository address>
```

for example, if we wanna clone a repository, from `https://github.com/azravian/python_exercise` we will use following command.

```
$ git clone https://github.com/azravian/python_exercise
```

now this remote repository will be downloaded in current root directory.

2. Make changes:

Now we'll do changes in the project files. let us create some files.

Create a file using command line.

In order to create a file in command line, we can use following command:

```
$ touch FirstFile.txt
```

it will create a *FirstFile.txt* in current directory.

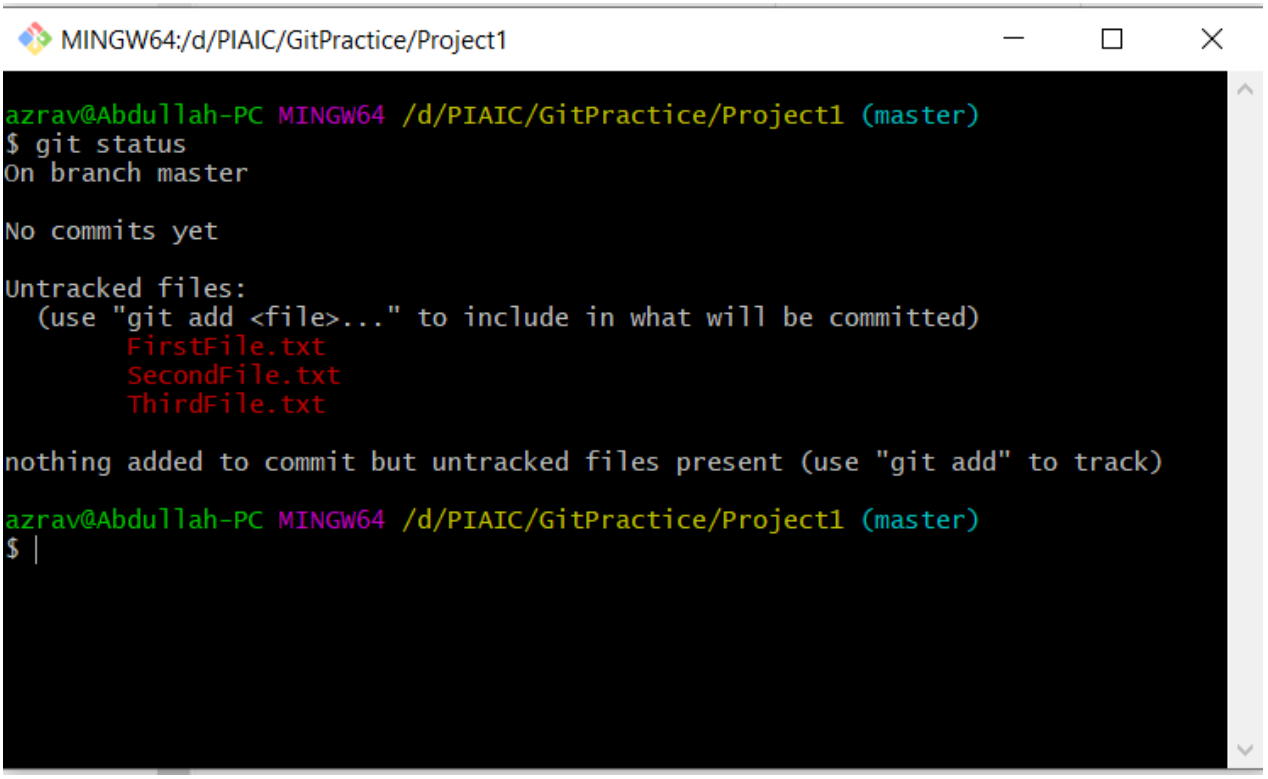
Now As we have created a file. we can now proceed with basic workflow of git VCS. let us create two more files named as SecondFile.txt and ThirdFile.txt.

Now we'll have 3 file in our root directory.

Change Text file:

let us make some changes in one file. we add some text in our *FirstFile.txt*. Now after changing file we can check the status for our file. it can be done by using following command:

```
$ git status
```



it shows the status of all files *untracked* So we need to stage our changes.

3. Stage Changes

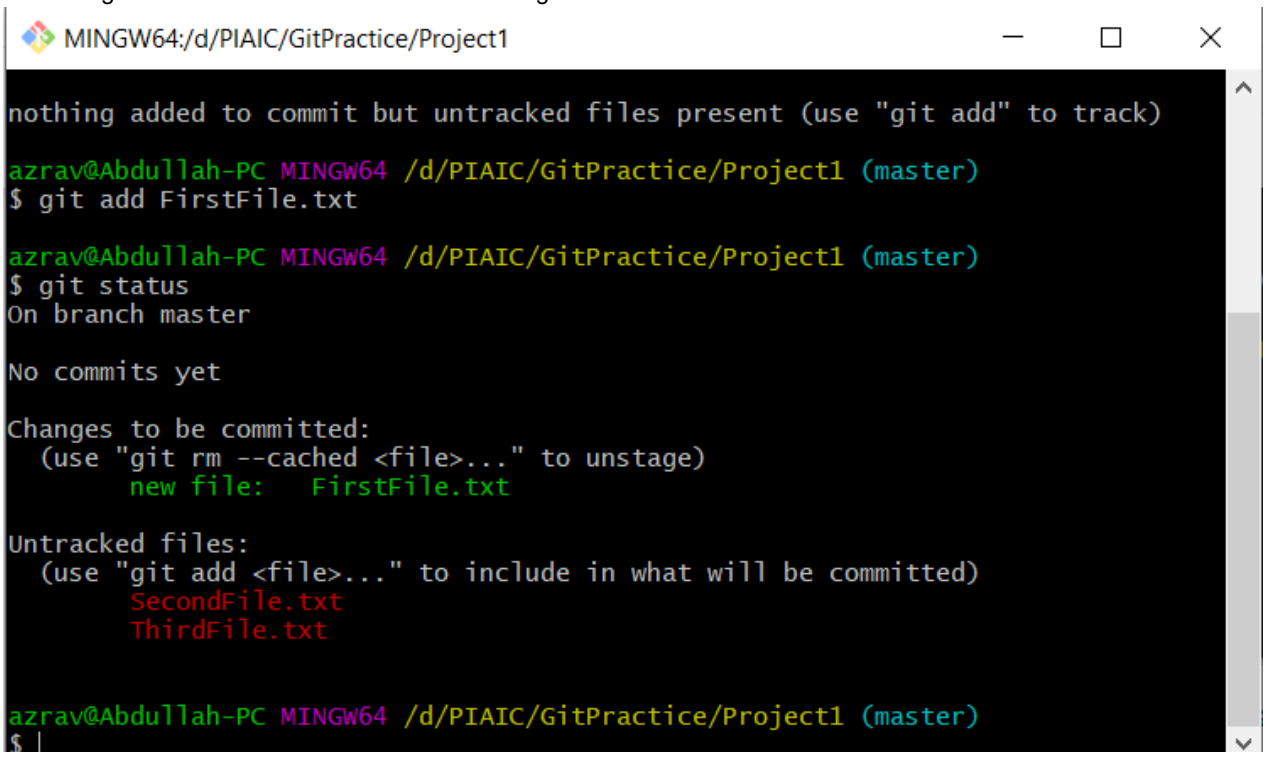
files can be staged for commit using following command.

```
$git add <filename>
```

For now, we only stage *FirstFile.txt*.

```
$ git add FirstFile.txt
```

after using this command let us check the status again.

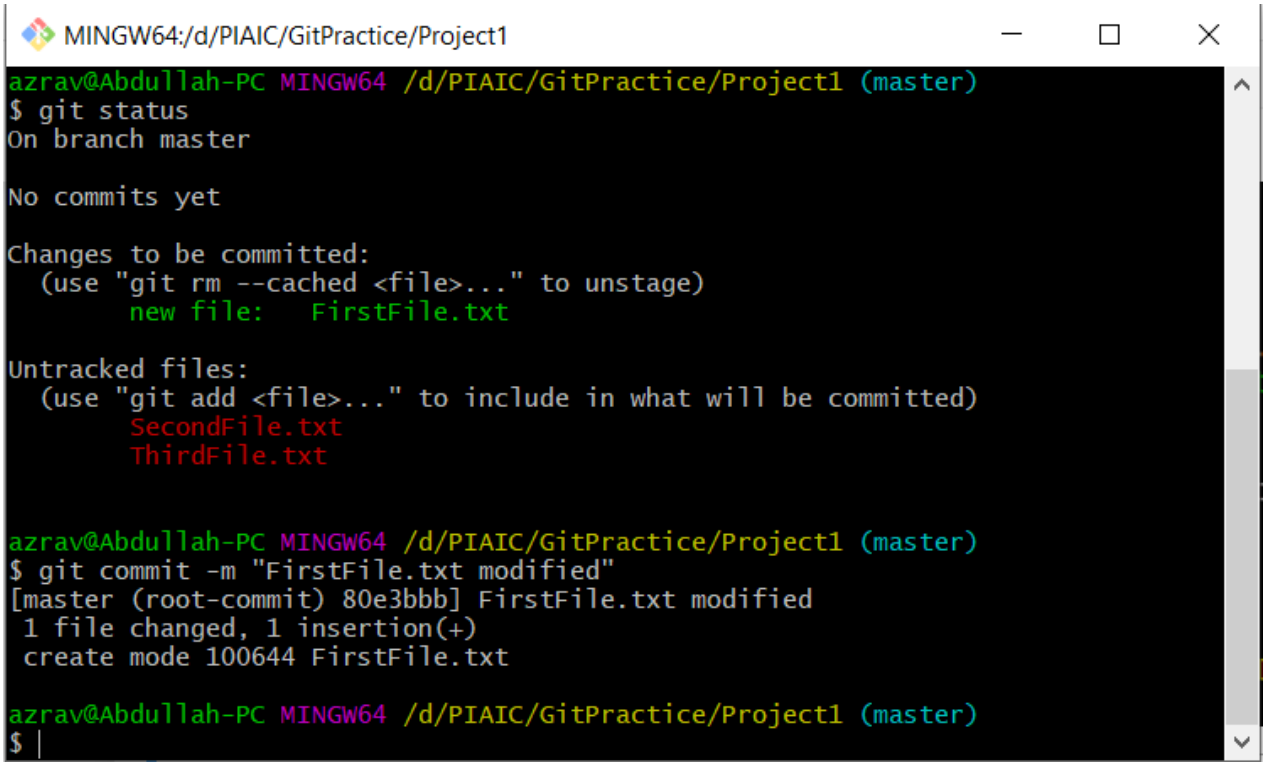


4. Commit changes.

Once we have finalized what changes in which file are final, we need to commit those changes. The commit is made by using follwoing command.

```
$ git commit -m "commit message"
```

by using `-m` we can specify our message for this commit. this message should carry some information about the changes in this commit.



Once commit made a *snapshot* of poject is created. The basic workflow of a local VCS completes here.

More on *Staging Area*

A good commit should only focus and about a single change as having multiple changes in single commit would be confusing. When we need to control our commits staging of file is pretty much useful as we can select the files and changes to be comitted. Now we'll look into more details into adding files into staging area.

`add <filename1> <filename2> .... <Filenamen>`

this command adds the specified file in staging area for commits. For example wewant to add our new files *SecondFile.txt* and *ThirdFile.txt* we can add these files to staging area using.

```
$ git add SecondFile.txt ThirdFile.txt
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        SecondFile.txt
        ThirdFile.txt

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

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git add SecondFile.txt ThirdFile.txt

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   SecondFile.txt
        new file:   ThirdFile.txt


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

Now both of these files have been added for commit, while our third file *ThirdFile.txt* is still untracked. filenames in git are case sensitive, it means *SecondFile.txt* and *secondfile.txt* are not same.

git add .

this command add all files in root directory into staging area. let us make changes in all files. Once all changes are made, run following command:

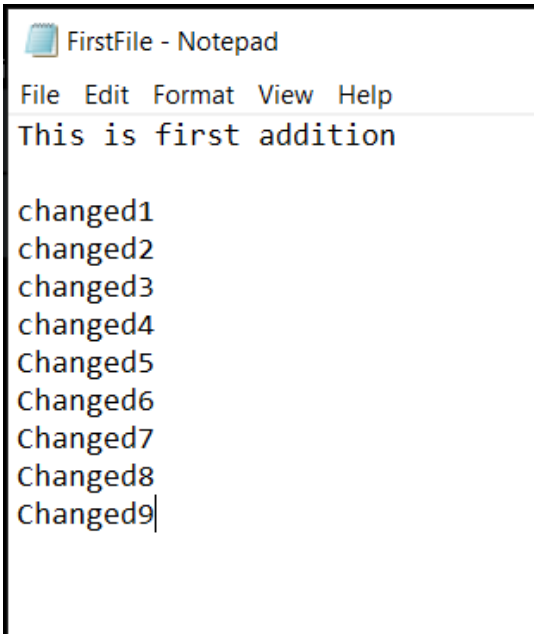
```
$ git add .
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   FirstFile.txt
        modified:   SecondFile.txt
        modified:   ThirdFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git add .

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   FirstFile.txt
        modified:   SecondFile.txt
        modified:   ThirdFile.txt


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

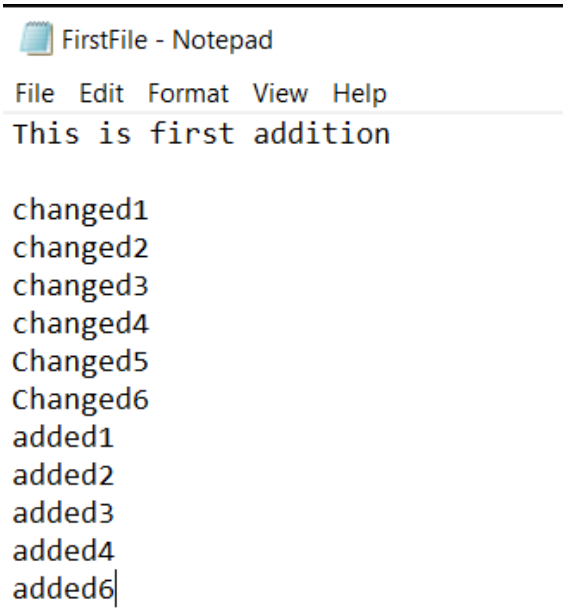
Notice that now at this stage all our files are tracked. and when `git status` is done, it shows all files have been modified.

`git add -p <filename>` or `git add --patch <filename>`

This command let us control the number of line in a file to staged for commit. For example, if I have deleted 4 line and added 3 line in same file and now I want to make separate commits for deletion and addition, I have to select which lines should be staged for commit. For example previosuly my *FirstFile.txt* have following content:



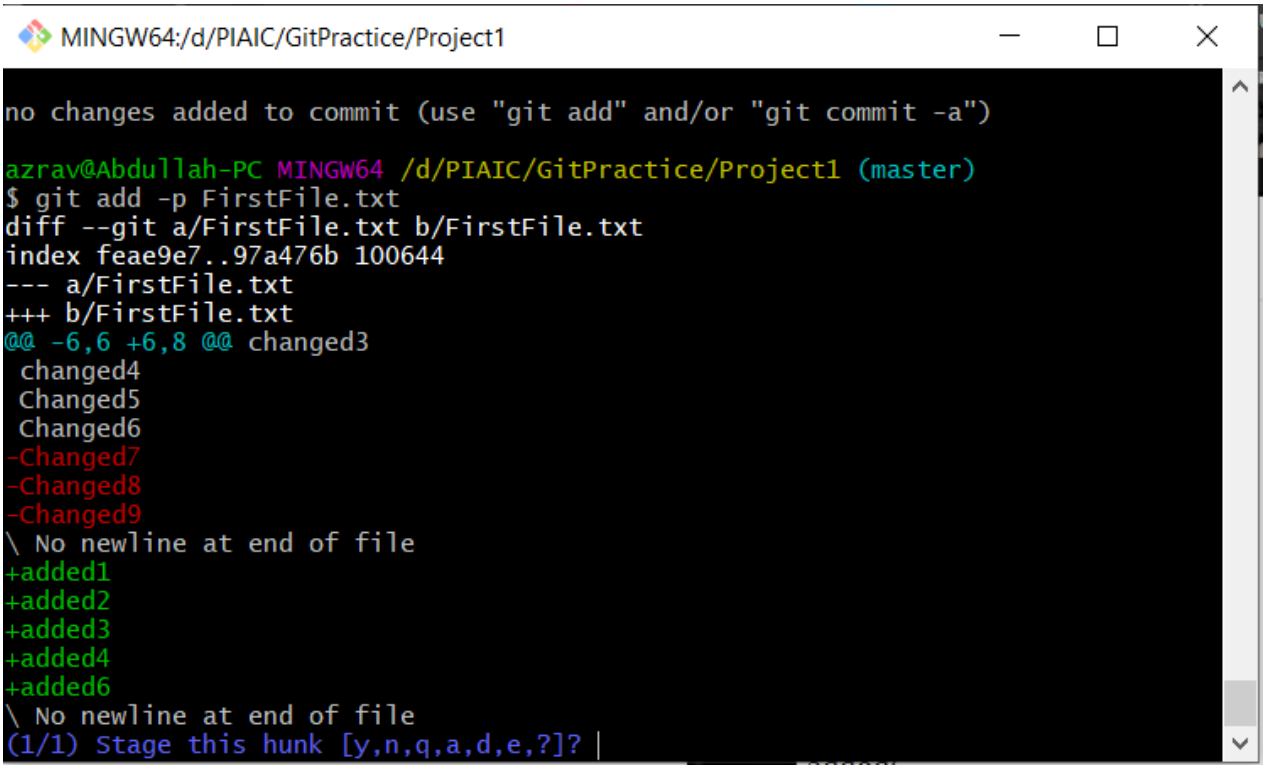
and after changes:



we have deleted some lines and then added some line. Now first we want to commit only the deletion and them addition. run following command:

```
$ git add -p FirstFile.txt
```

it will open following screen. and before going into more details it must be added that a change is called **hunk** in git. like the addition we have made is one *hunk*



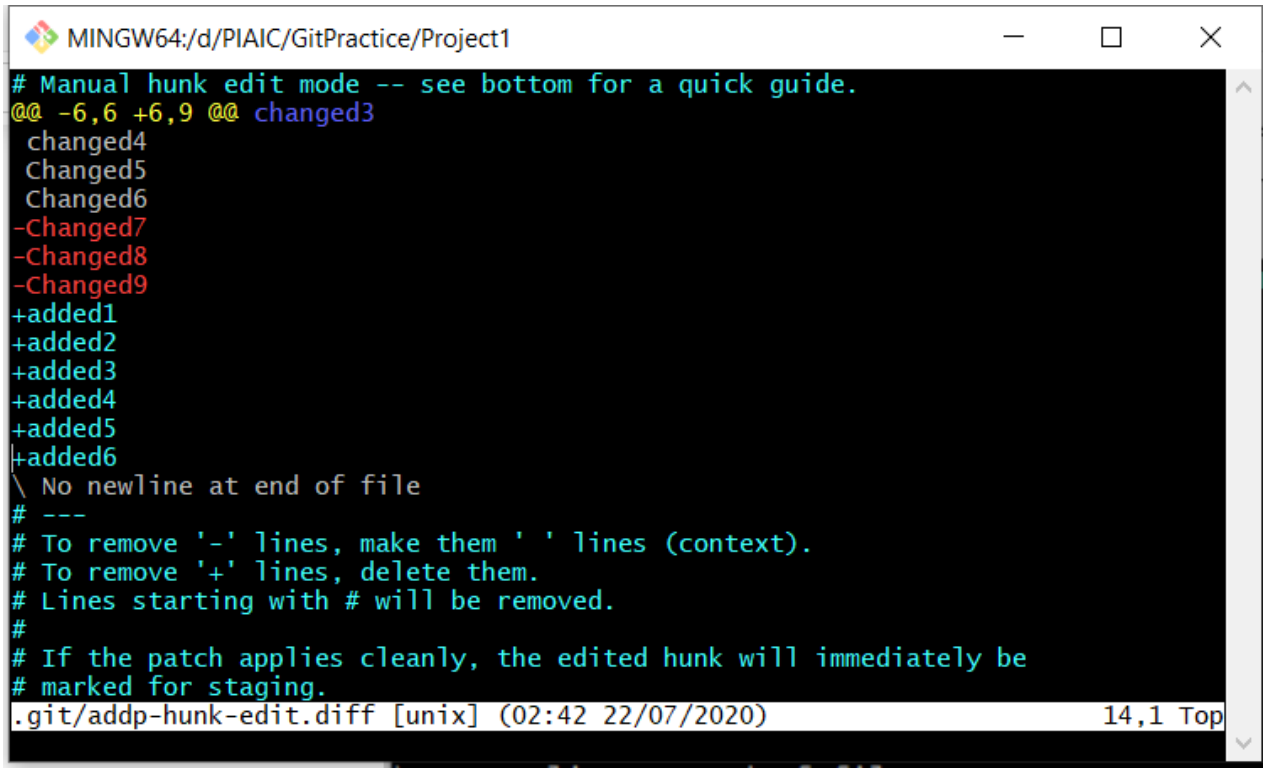
In the end it is showing some options:

y - stage this hunk - this command adds the current hunk to staging meaning it is ready.
n - do not stage this hunk - this command won't add the current hunk to staging.
q - quit; do not stage this hunk or any of the remaining ones - this command quits out of the staging process. Any hunks before this one that has been added to staging will still be staged but the current hunk and all hunks after will be ignored.
a - stage this hunk and all later hunks in the file - this command works similar to the `git add .` call. It will stage all changes made from this hunk on.
d - do not stage this hunk or any of the later hunks in the file - this command will not stage this hunk or any hunks in the same file.
e - manually edit the current hunk- this command opens vim and allow you to edit the hunk of code in your terminal.
? - print help - this opens up the menu above.

One option not printed here is:
s - split this hunk - this option is only available if the current hunk of code has an unchanged line of code between edits. This will split the hunk into two separate hunks allowing you to stage them individually.

Editing hunk:

Now we need to edit our hunk to commit only deletion. at this stage, enter `e` and press enter, it will enter in *manual hunk edit mode* in *vim*.

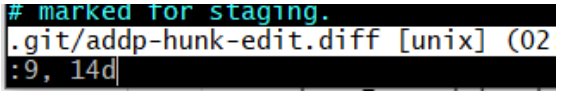


in this mode we can see our changes. The line with red txt starting with a `-` are removed and lines with blue text starting with `+` are the addition.

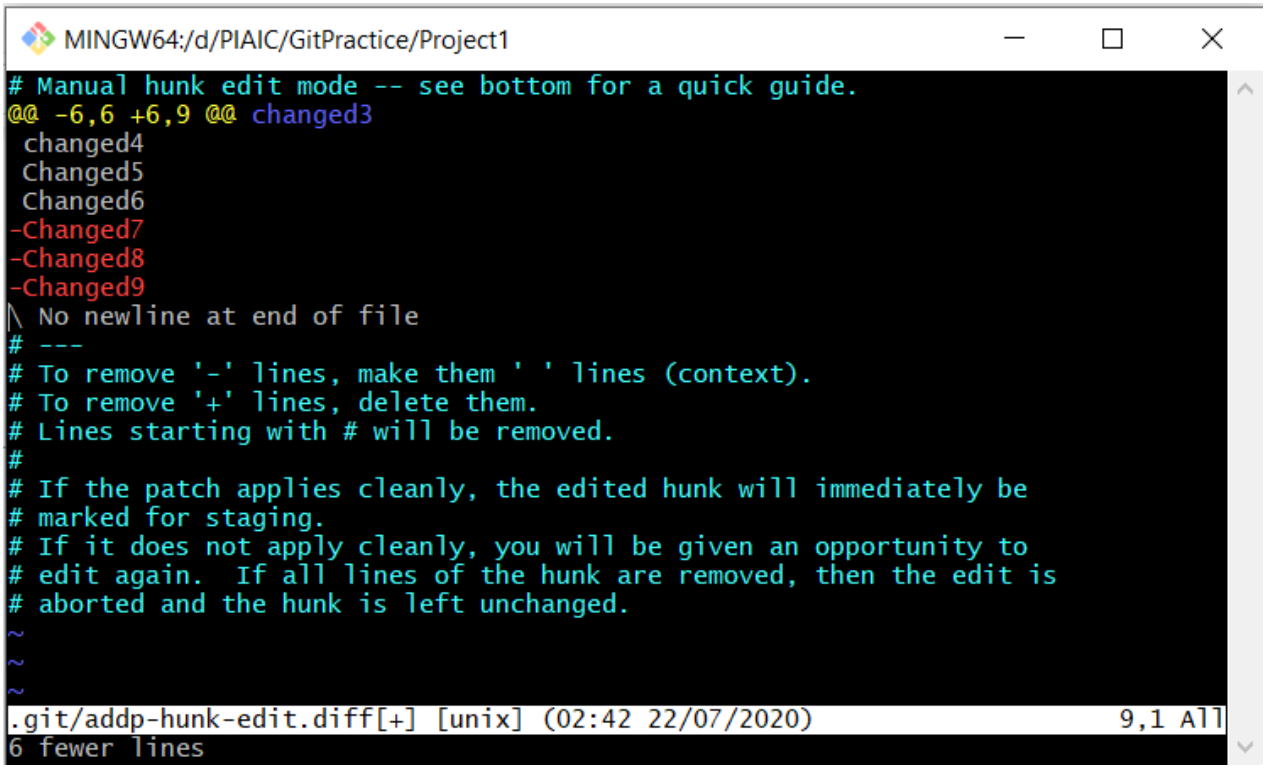
1. if we want to exclude addition from staging, we need to delete these line starting with a `+`.
2. if we want to exclude deletion from staging, we need to replace a `-` with ` `. (A blank space)

Excluding addition from staging:

In order to delete, move cursor to the line we want to delete and press `d`, the line will be deleted. It we want to delete multiple line we need to use this syntax: `:NumberOfStartLine, NumberOfEndLine d` and press `Enter`. In our example, I need to delete line 9 to 14, so I need to enter `:9, 14d`.

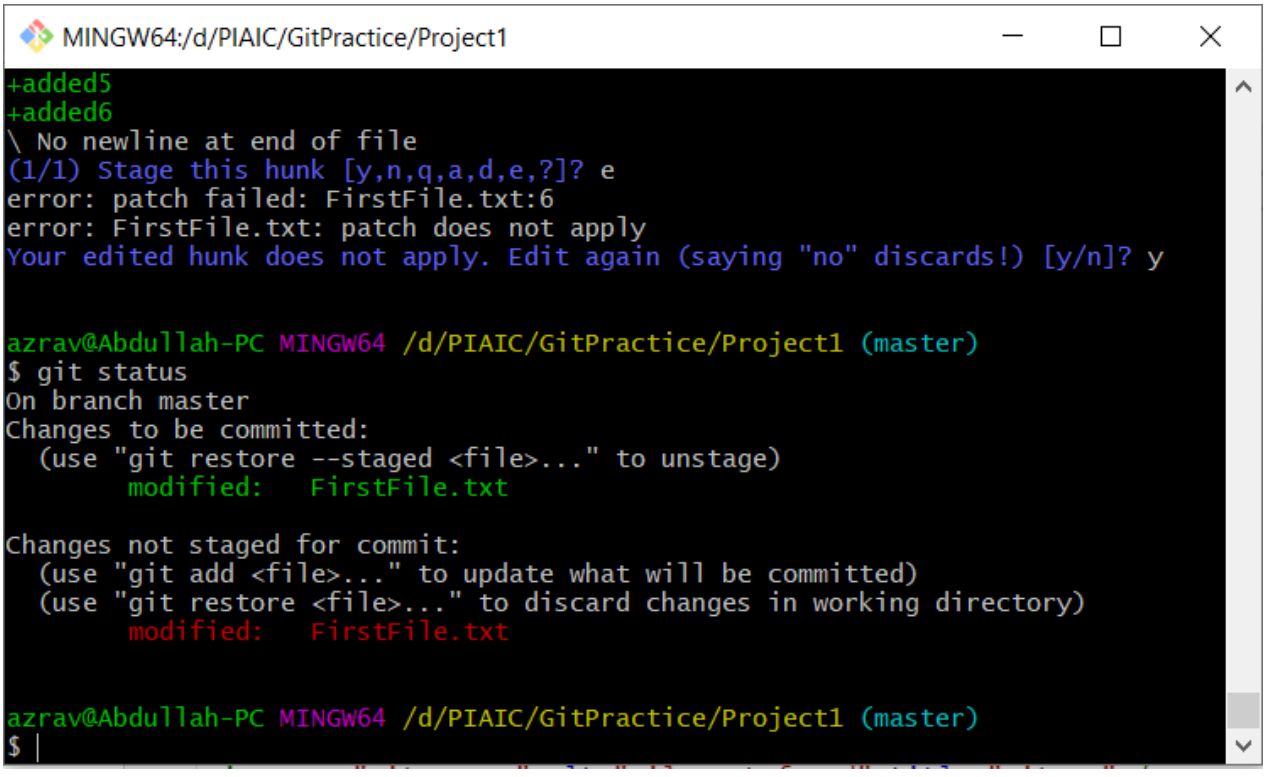


Once run this command, all these lines will be deleted.



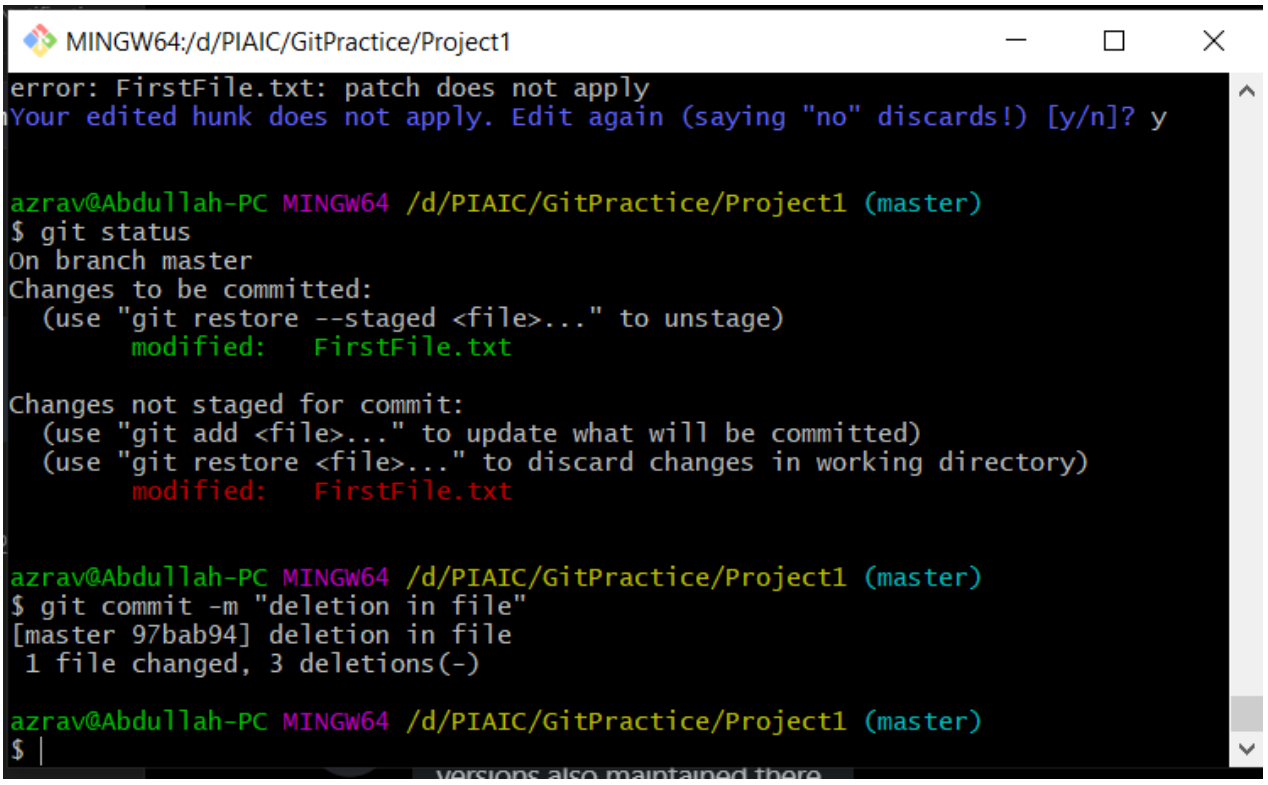
Now we only have line which are removed. Now in exit *vim* with `:wq` command. Now our deletion of line is staged for commit.

Checking status again:



we have same file staged and unstaged for commit, it's because we have staged the deletion in file to be commited and and addtion not be commited. now we commit our deletion:

```
$ git commit -m "deletion in file."
```

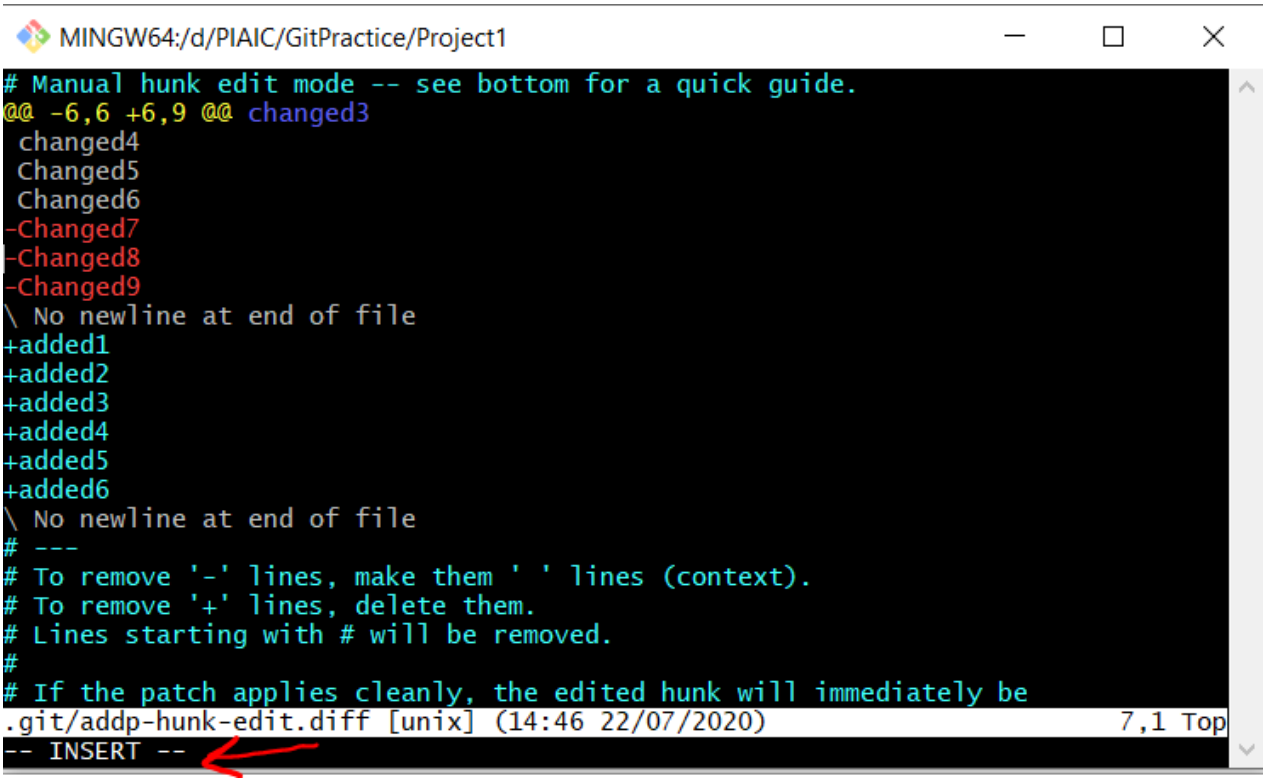


as we can see from commit details, only 3 deletions were commited. and after that only addition is left which we can stage and commit as:

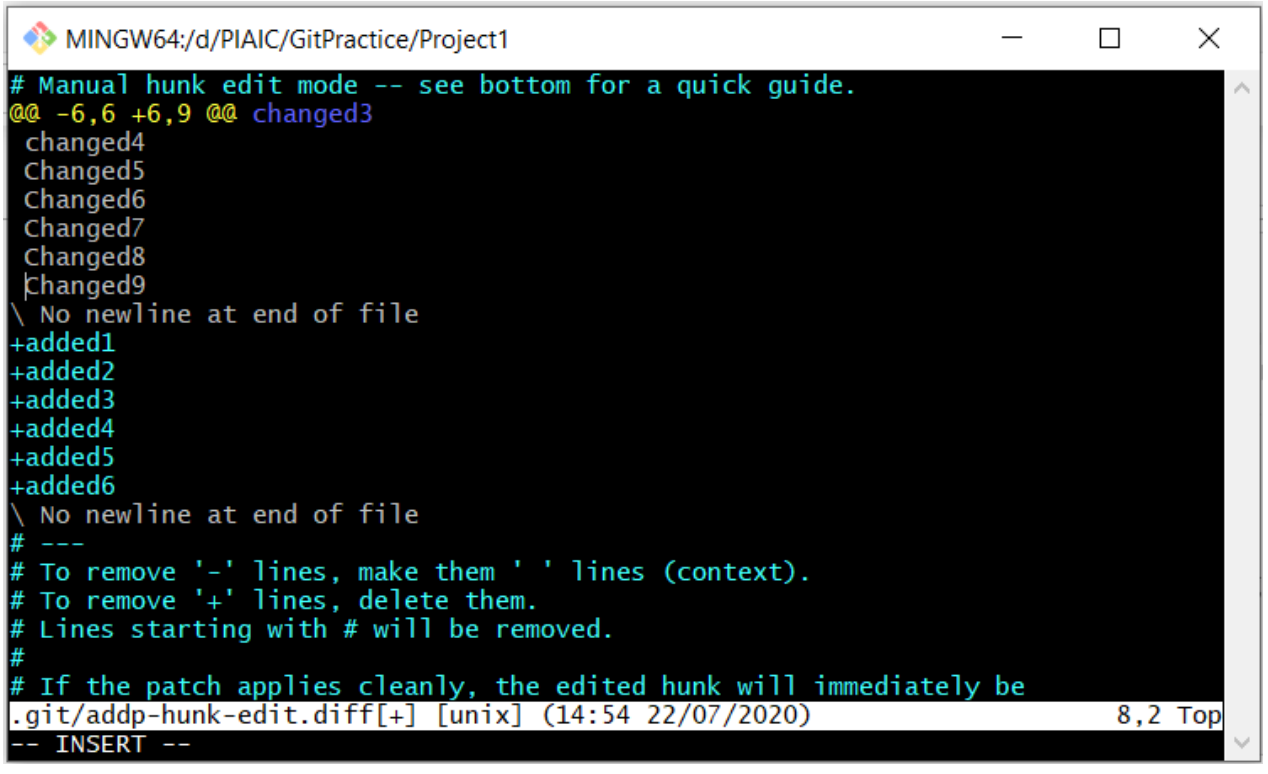
```
$ git add FirstFile.txt
$ git commit -m "Addtition in FirstFile.txt"
```

exclude deletion from staging

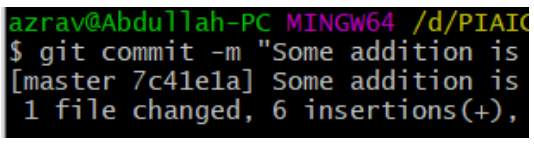
I previous example, we have removed additions from being staged and staged our deletions. In case we need to satge only additions and not deletion, it has a different procedure. Let us make some changes and again open in `manually hunk edit mode`, now press `i`, and in the bottom left corner insert will be written, then press enter and we will enter in edit mode:



Once in edit mode replace `-` with a ` ` (a blank space) before the line which are to be removed from staging:



Now press `esc` key to exit from *insert mode* and enter `:wq` to exit *vim*. After commit, we can see only additions are commits.



`git add *.ext`

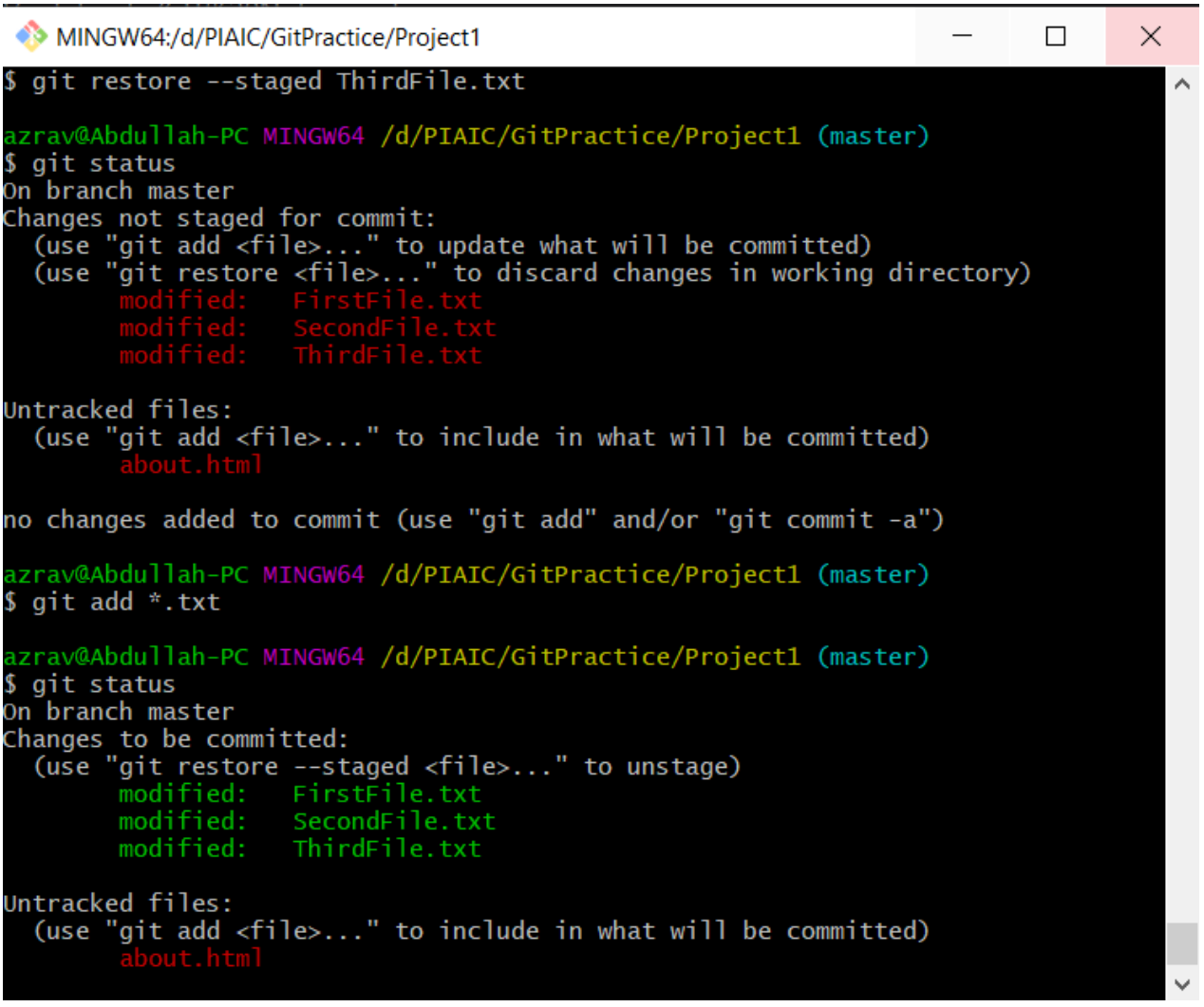
At this stage let us add some file other a *txt* file. For example, I have added an html file in my directory. If all file of one type, i.e. with same extension, are required to be staged, we can use following commamd:

```
$ git add *.ext
```

where *ext* is the extension of our required files. For example, in our example, we need to commit only txt files.

```
$ git add *.txt
```

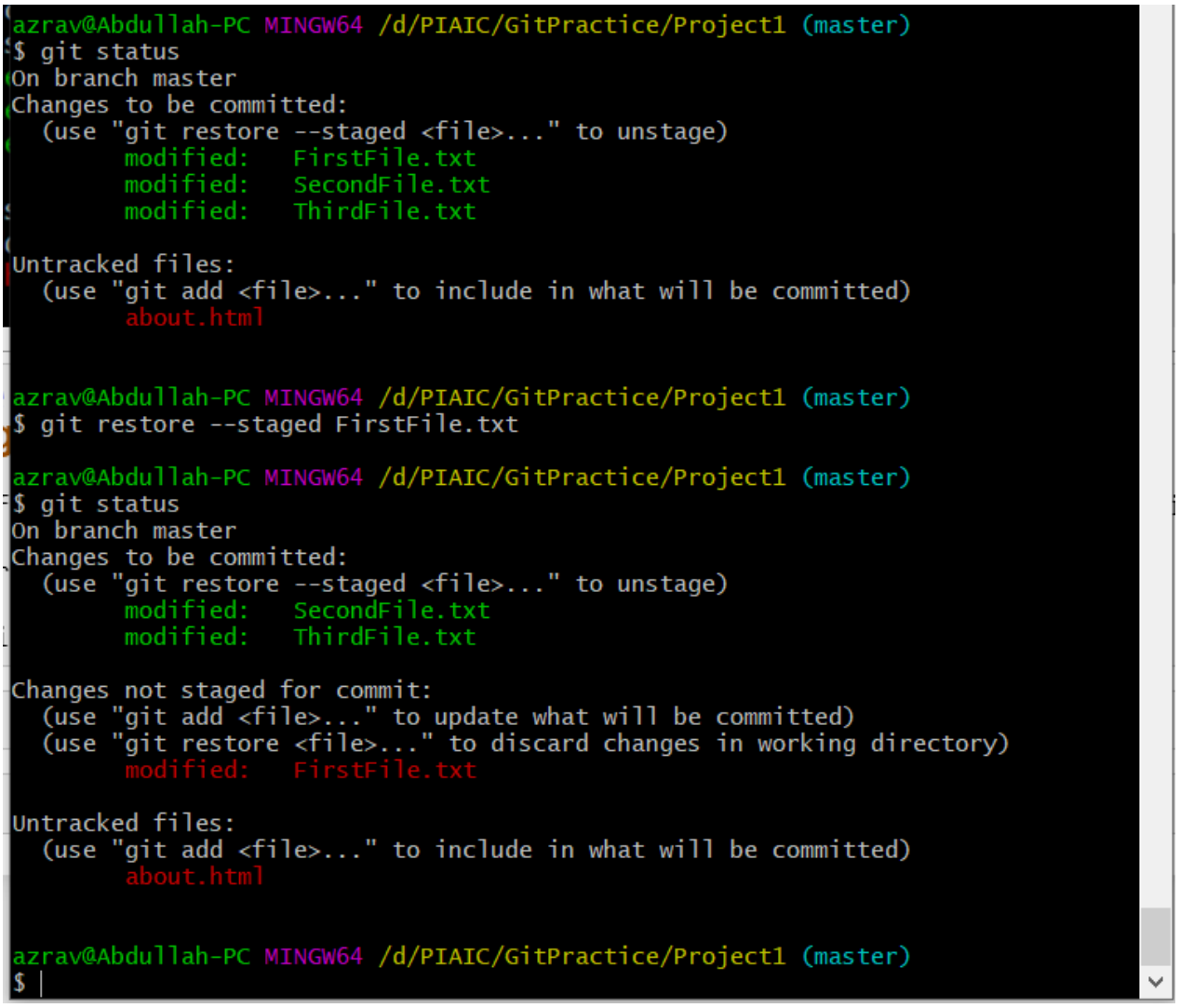
this command will add all file with *.txt* extension to staging area.



Removing a file from staging area.

`git restore --staged <filename>`

In case we have stage some file mistakenly, and now we need to remove it from stage area. In our example if we wanna remove FirstFile.txt, we'll use: `$ git restore --staged FirstFile.txt`

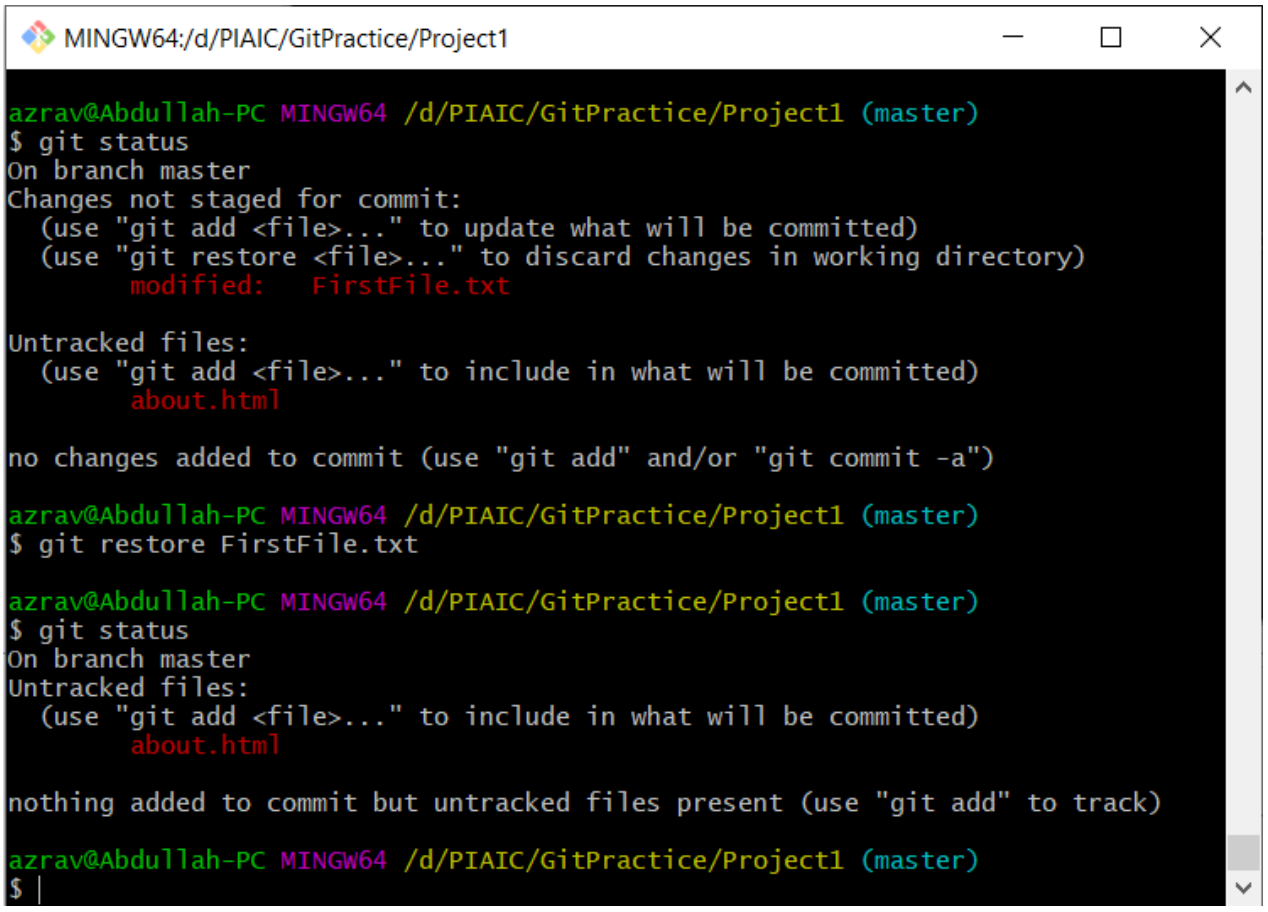


Discard changes in file.

`git restore <filename>`

Should one needs to restore his file by discarding all changes in some file, we'll use this command. However this file shouldn't be staged and is tracked. For example, chagnes have been made in *FirstFile.txt* and then want to discard these changes, we will execute following command:

```
$ git restore FirstFile.txt
```

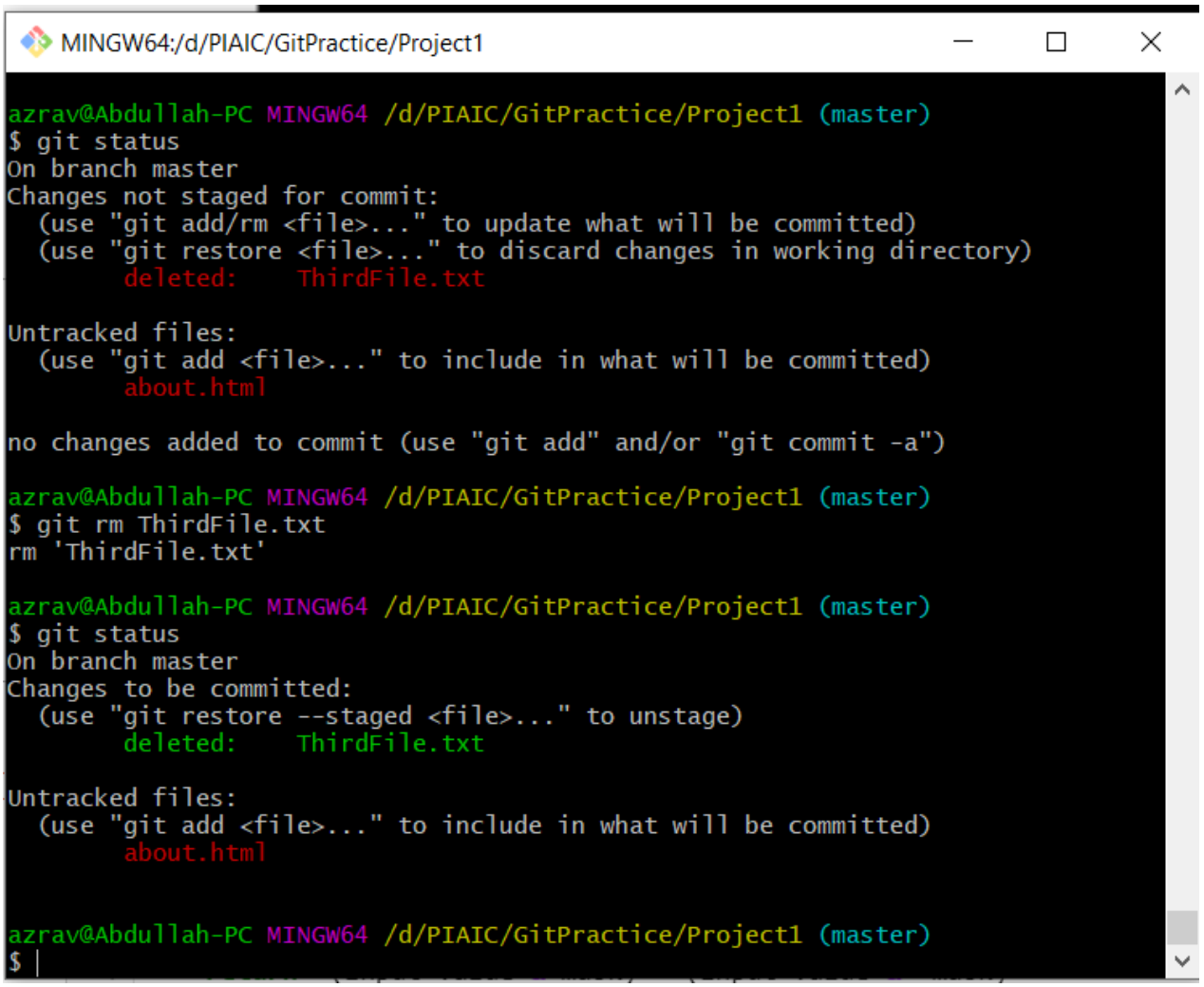


if we open *FirstFile.txt* we'll see all changes were discarded.

Removal of deleted file

`git rm <filename>`

In case we have deleted a file from our working directory, we need to stage this deletion of file for commit. Let us delete a file, say *ThirdFile.txt*, in our working directory which was being tracked in git. We can make this deletion ready for commit using following command: `$ git rm ThirdFile.txt`



Now our deletion is staged for commit.

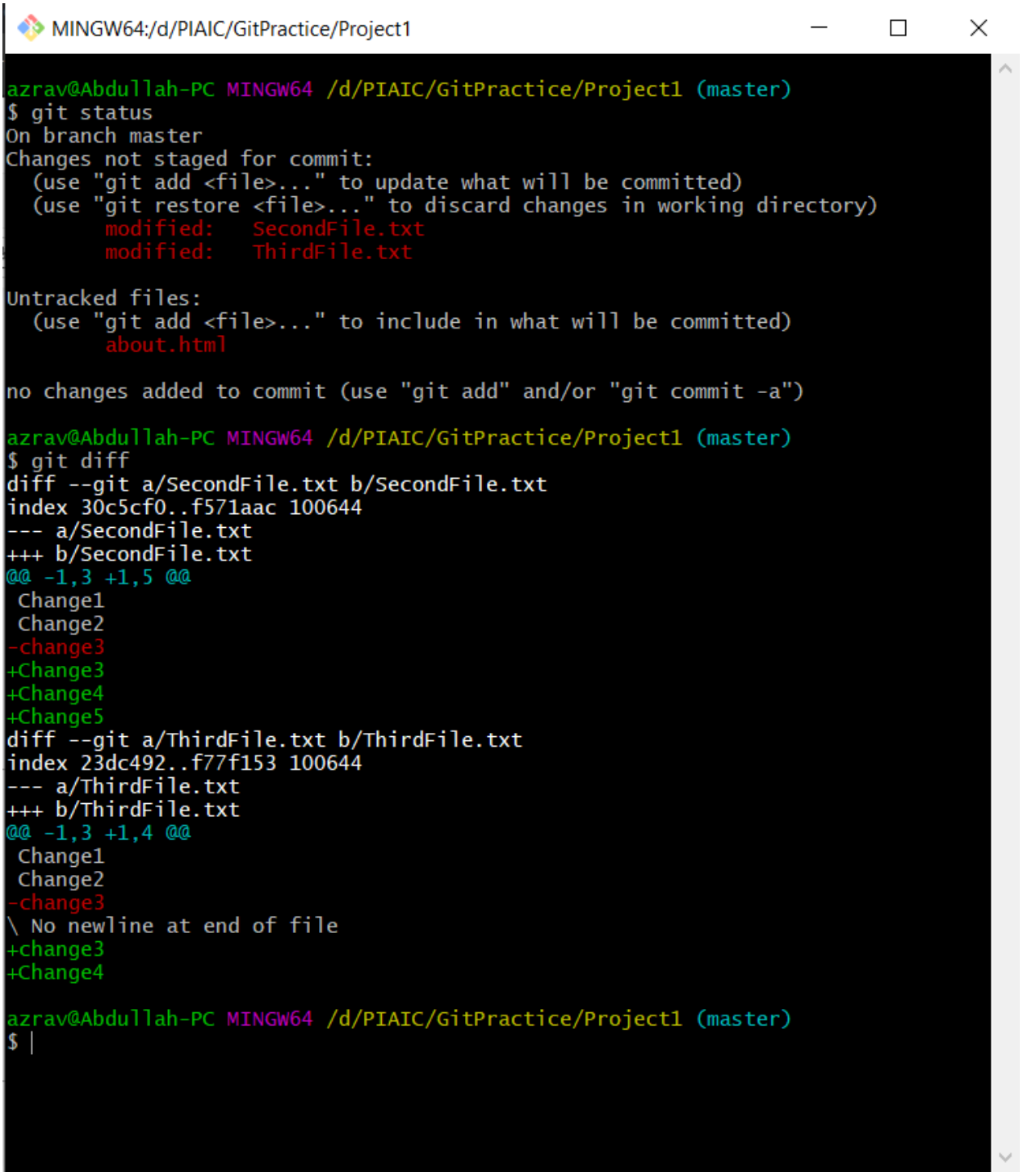
Check diffrences in tracked file.

`git diff`

Let us make some changes in *SecondFile.txt* and *ThirdFile.txt*. if we wish to see changes in these file, we would run following command:

```
$ git diff
```

it will show all changes made to both files.



and in case we need to see changes in one file only, we will use following command:

```
$ git diff <filename>
```

Creating a commit

Once done with staging we need to commit changes. A good commit should be:

1. A good commit should include changes related to one feature or issue.
2. It should have short description about the change.
3. Changes must be test and final.

Let us see more about creating commit:

git commit -m "Message":

This command commits the staged changes. `-m` flag is short for `--message` whcih specify a short a message to commit. Now suppose we have made some changes in *FirstFile.txt* by adding two lines in the last. We can commit by using:

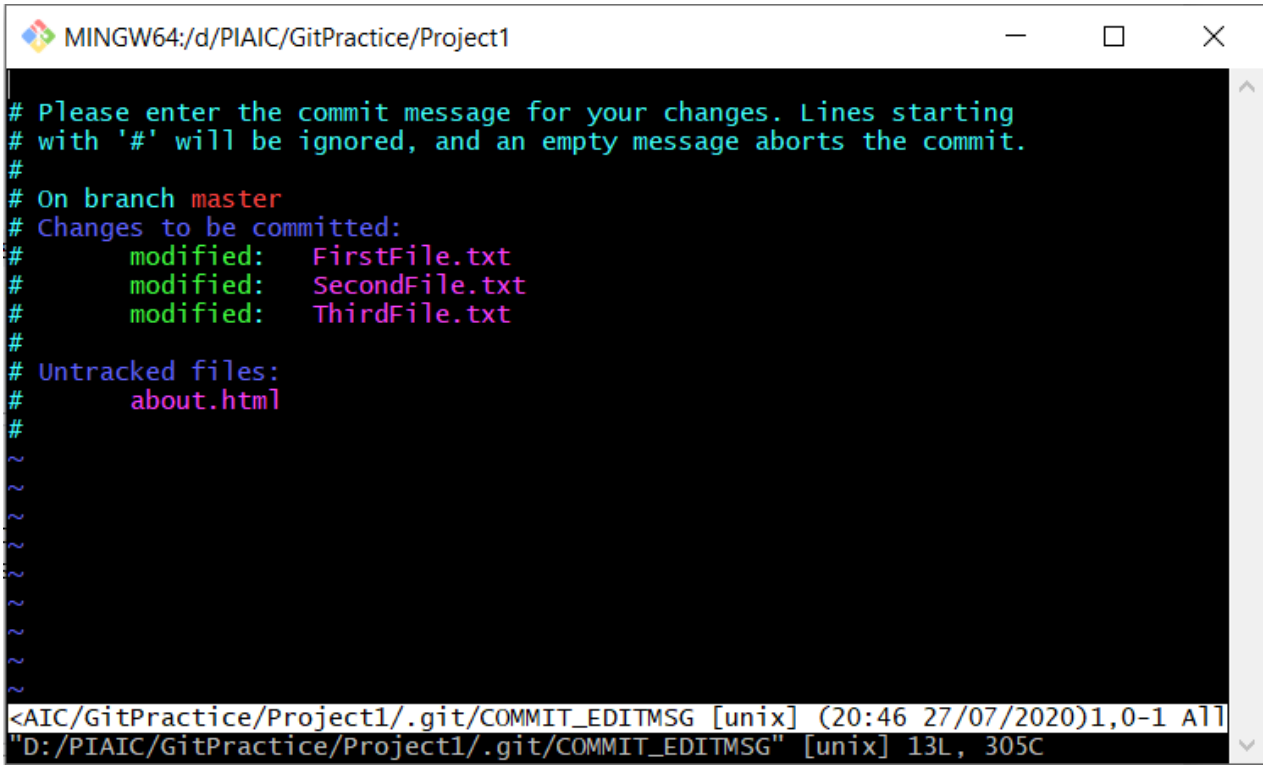
```
$ git commit -m "Added last two lines"
```

git commit

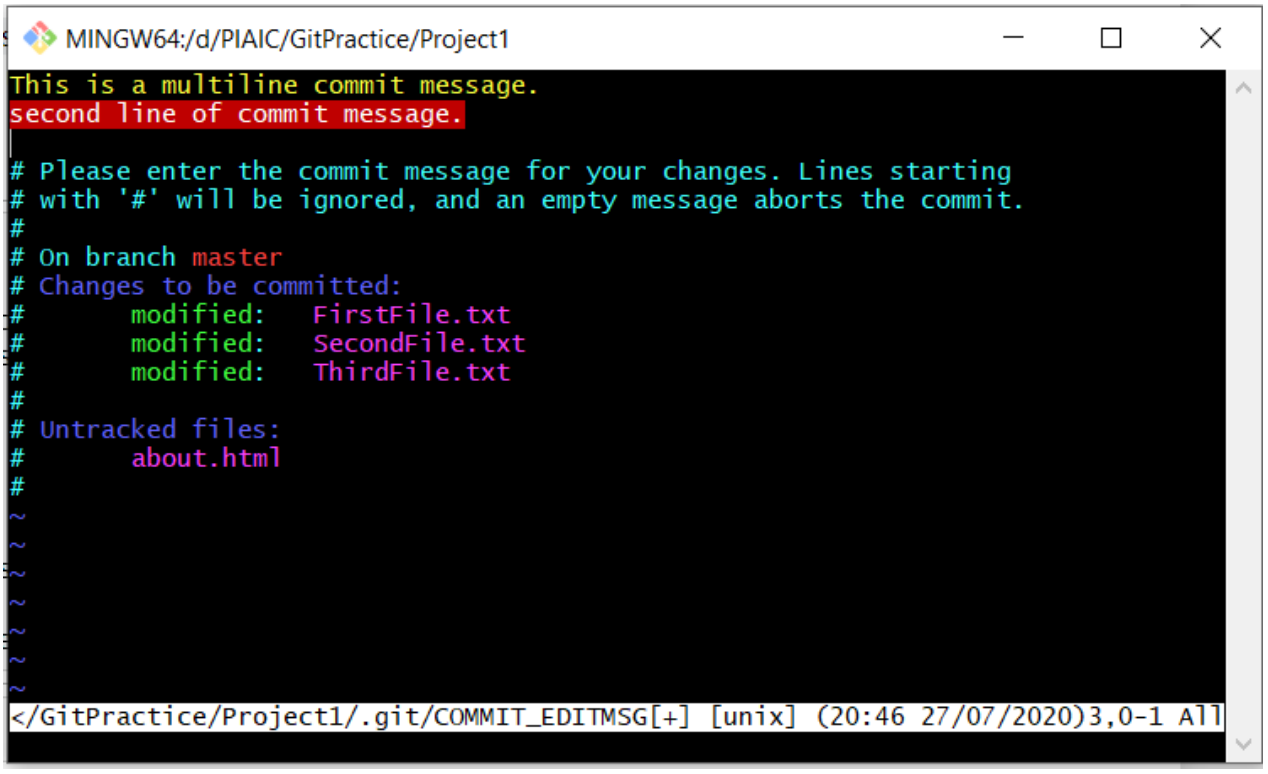
This command is used for creating multi line message for a commit. When this command is entered, git command line enters in *vim*. For example, after changing our text files and staging these changes, enter following command.

```
$ git commit
```

following screen will open:



We can see the changes files. Now simply type commit message.



once the message is entered press `esc` key and then type `:wq` and and press `enter`

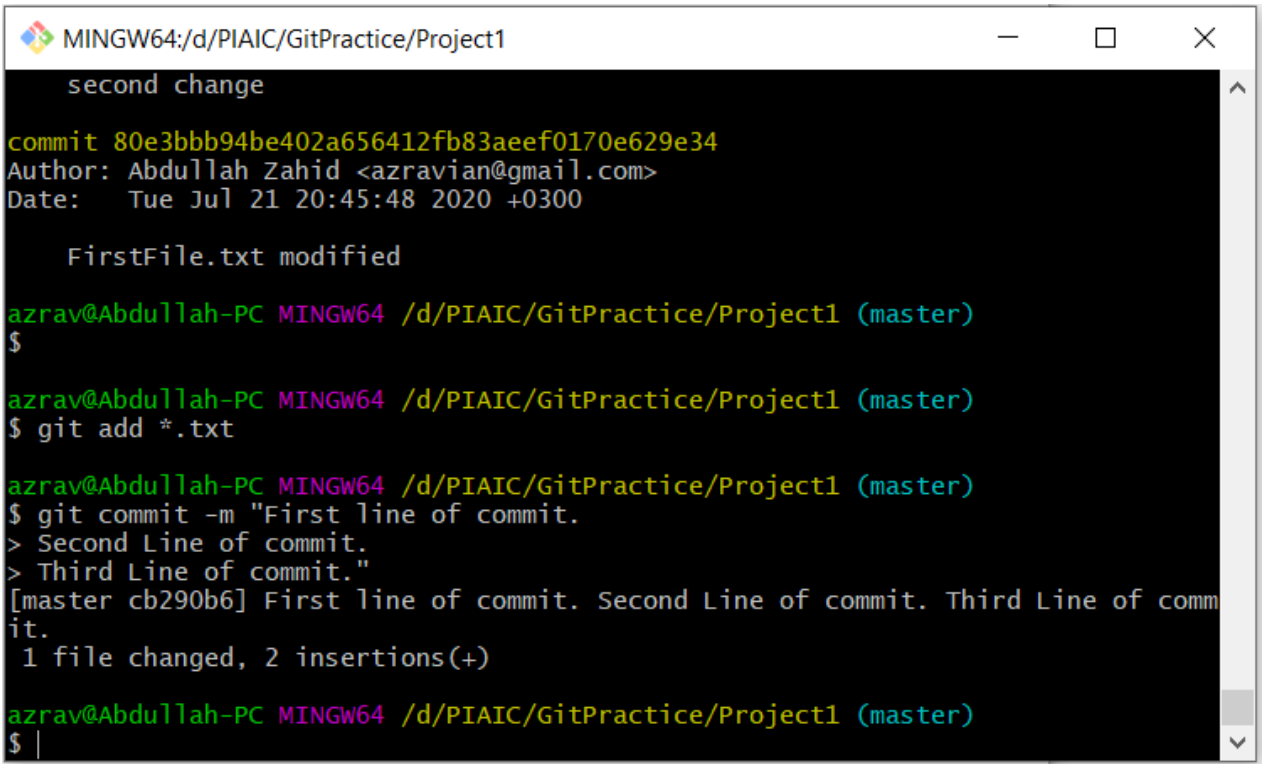
Multiline commit using `-m` flag

if one need to enter a multiline commit message using `-m` flag, one need to use following procedure

1. Type `git commit -m "First Line of commit.` and leave the second `"` at the press `enter`.
2. Type scond line of message.
3. Keep typing all line.
4. Once all line are typed add `"` to the end of last line and press `enter`. for example:

```
$ git commit -m "First line of commit.
> Second Line of commit.
> Third Line of commit."
```

Now this commit will have three lines.



git commit -a

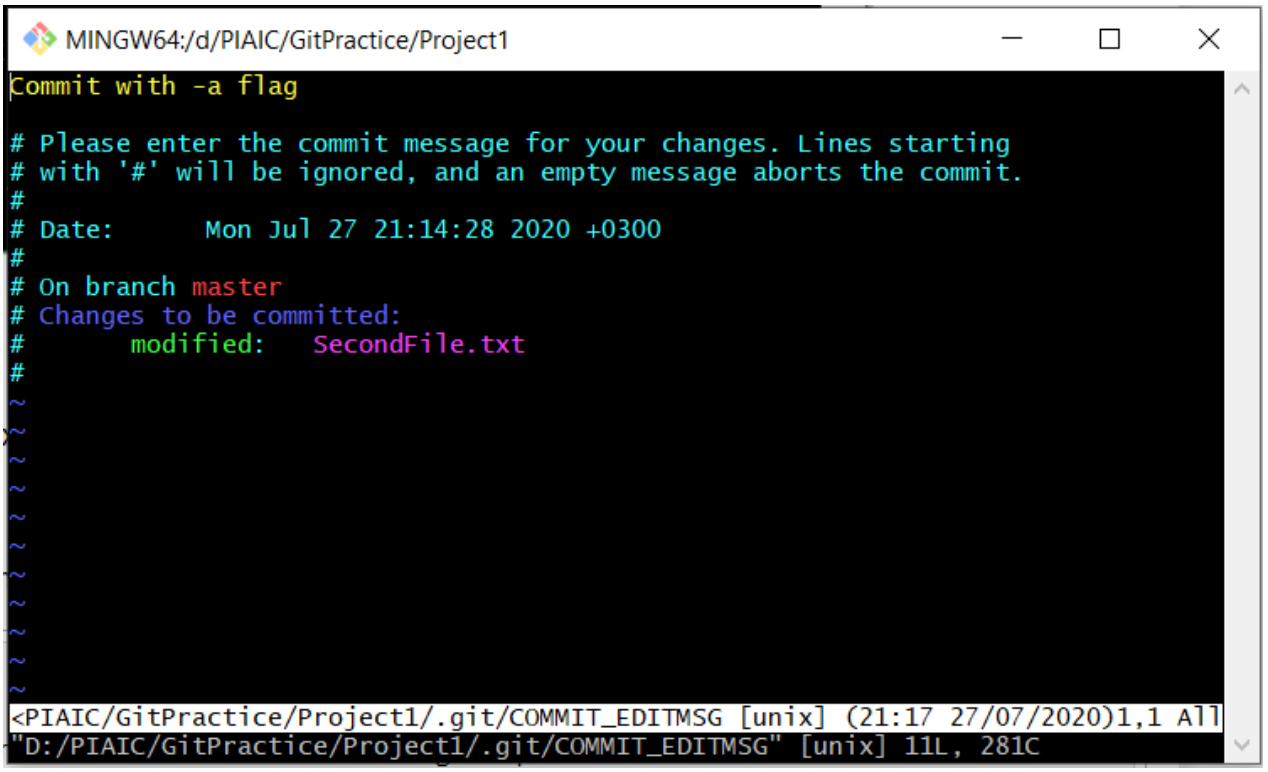
this command commits all change in tracked file, even if not staged. for example we wanna commit with `-a` flag:

```
$ git commit -a -m "message"
```

We won't need to stage our changes, all changes in track files will be commited.

git commit --amend

this command is used to change the last commit. when this command is entered the git opens *vim*.

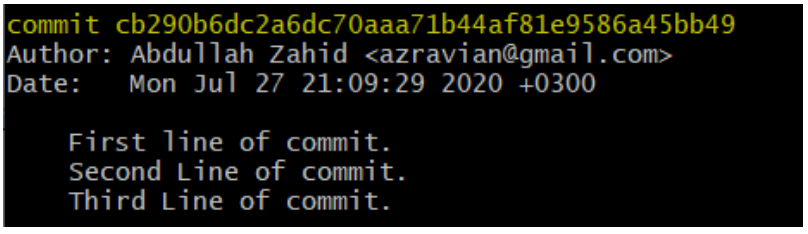


We can change our message, then `write and exit` from *vim*

commit logs

`git logs`

This command shows history of all commits in chronological order with the most recent commit first. This log look like as below:




```
commit cb290b6dc2a6dc70aaa71b44af81e9586a45bb49
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Mon Jul 27 21:09:29 2020 +0300

    First line of commit.
    Second Line of commit.
    Third Line of commit.
```

Each entry have following informations:

1. commit hash:

A unique 40 character ID assigned to each commit. This is shown in first line

2. commit Author:

Shows user name and email of the user who created this commit.

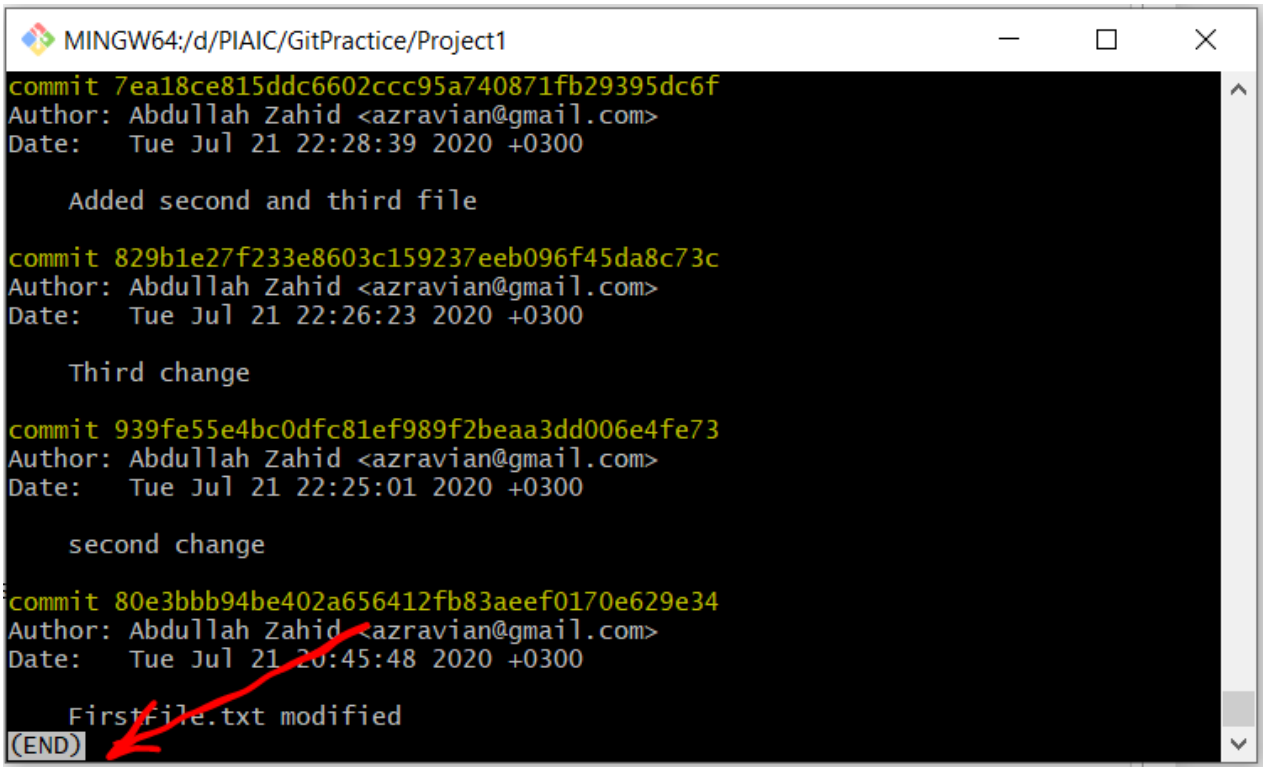
3. commit Date:

line 3 have date and time the commit was made.

4. commit message:

The following line contains commit messages.

If all log can not be populated on a single page, they are distributed over multiple pages and in such case a `:` is displayed at the bottom left corner of CLI. Jutst press `space` to move to next page. When all commits are displayed, `(END)` appears on the bottom left corner.



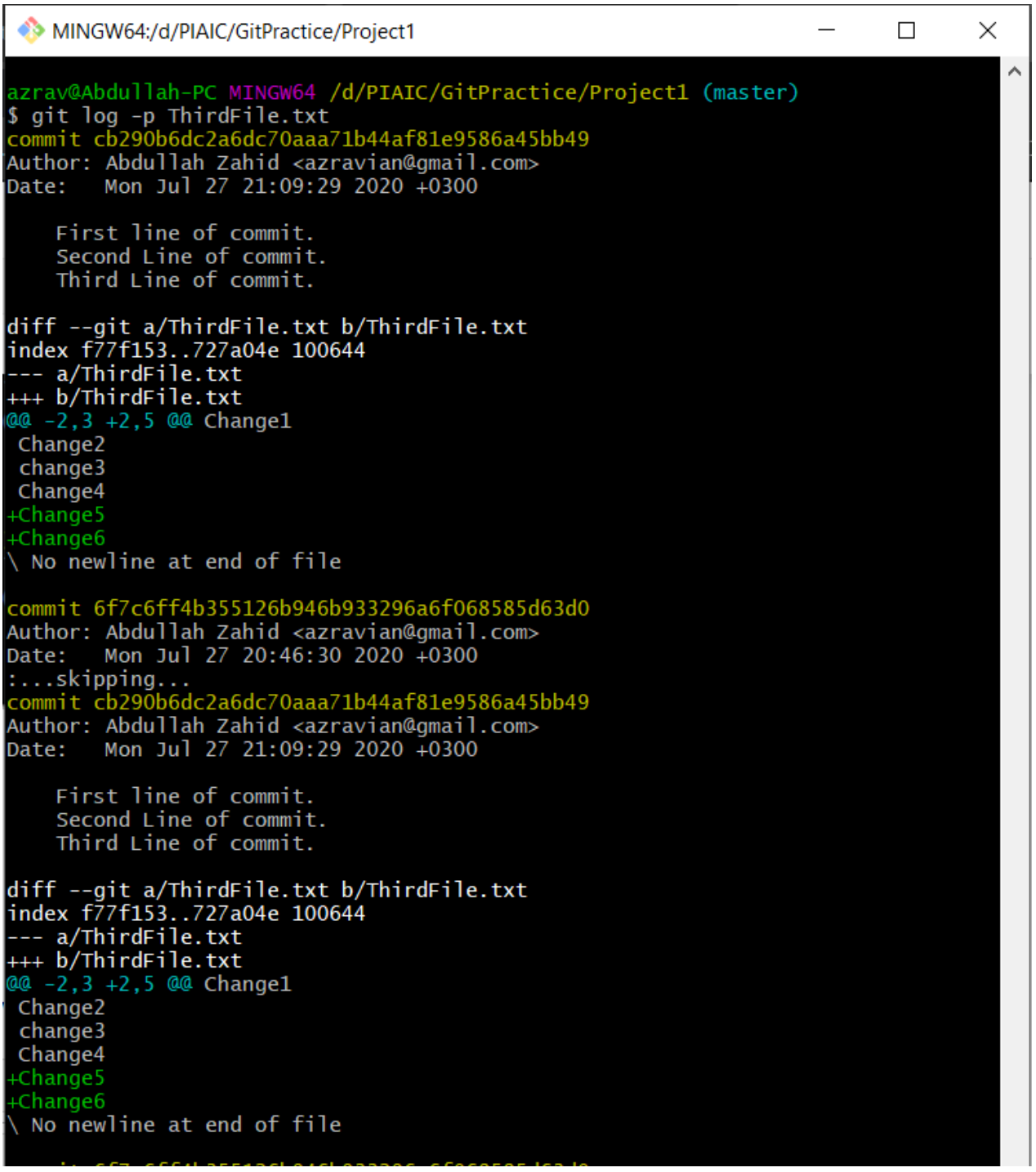
In order to exit, press `q`

detailed log of commits in a file.

`git log -p <filename>`

This command shows the commits associated to a given file. It also show the details of changes made in file. For example, following command will show all changes and commits in *ThirdFile.txt*:

```
$ git log -p ThirdFile.txt
```

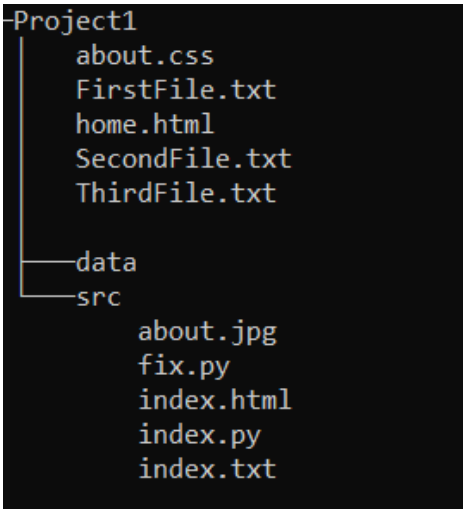


Ignoring files:

In order to ignore files, open a text editor, for example notepad in windows and save file as `.gitignore` in working directory. (if using Mac add `.DS_Store`). Before jumping in we will see our files in directory.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/project1 (master)
$ ls
about.css  data/  FirstFile.txt  home.html  SecondFile.txt  src/  ThirdFile.txt

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/project1 (master)
$ cd src

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/project1/src (master)
$ ls
about.jpg  fix.py  index.html  index.py  index.txt
```

In directory Tree:



Ignore a specific file:

if we wanna ignore a specific file, just add the name of file with extension to this file. For example, we want to ignore a file name `about.html` just add this name to add this filename to `.gitignore` file.

Ignore all files in a folder:

When we want to ignore a folder, we need to put the path address of this folder into this file.

Ignore all file of one type:

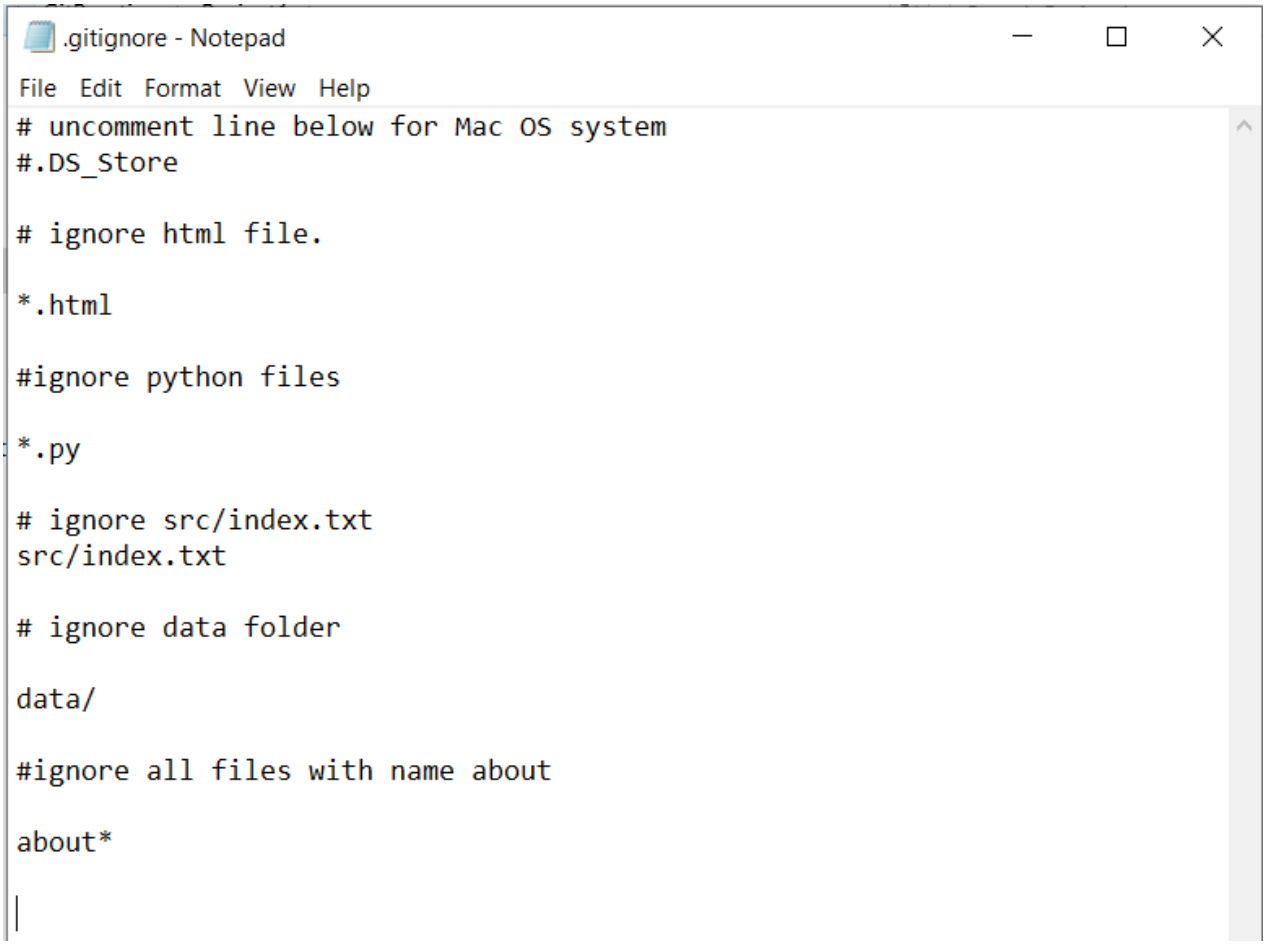
If we want to ignore all files of a type, we need to add `*.ext` to this file where *ext* is the extension of file. For example, python files have *.py* extension. If we add `*.py` in this `.gitignore` file all python file, be it fix.py or index.py, will be ignored

Ignore all file with same name:

If we need to ignore all files with a name we just add file name to this file and appending a `*` in the end. All files of this name irrespective of there type with this name will be ignored. For example, `about*` will all files named as about

comments in `.gitignore` :

in order to create a comment in this file, we start comment line with `#` .



Now we added files and folder names in `.gitignore` file. and when we check status, we can that none of these file appeared in status.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/project1 (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)
```

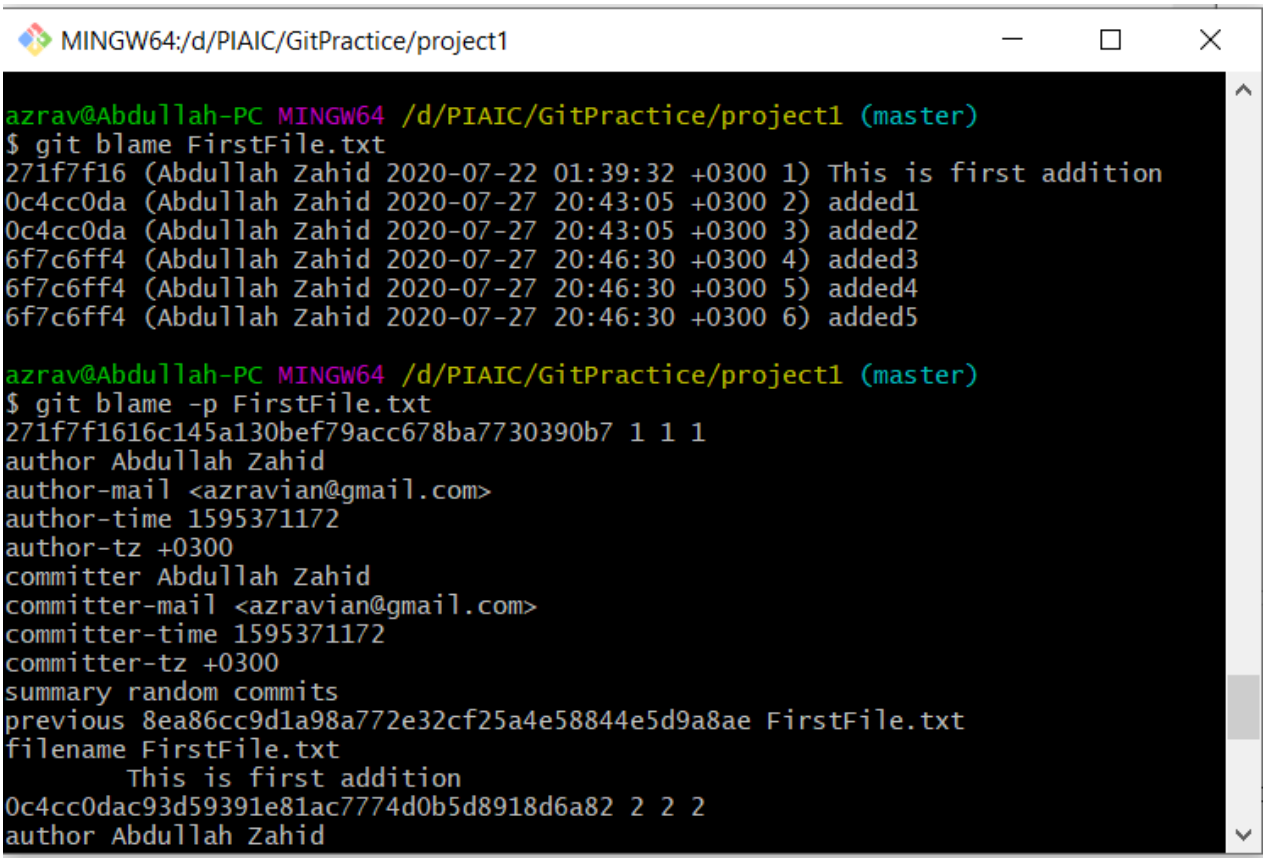
Check Who commited what and when:

`git blame <filename>`

This command shows what commits were made in a file, who made them and when. It just shows first 7 character of commit hash.

`git blame -p <filename>`

same as `git blame <Filename>` but it shows more details. The difference of results can be seen from the image below:



Branching

Bracnhes makes it easier to work on multiple contexts of a project. This context can be a new feature, bugfixe etc. When we commits changes in one branch, these changes are limited to one branch and other branches are left unchanged.

When Working on git we are always working with branches. The current branch we are working on is called `active` or `checked out` branch. A very common term used in git is `Head`. *Head* points to the last commit of current active branch.

Seeing current position of head.

When we use `git status`, the first line shows the current position of our head. For example, first line reads `on branch master`, which means our head is currently on *Master* Branch

```
$ git status
On branch master
nothing to commit, working tree clean
```

Creating a Branch

`git branch <branch name>`

This branch would create a branch in repository. It is a good idea to name a branch such that it reflects the context it is made for. For example, If we need to work on *FAQ* page, we may consider it naming `FAQ_Branch`. Let us create a branch named `fourthfile-branch` as we would use this branch for our new file named, *FourthFile.txt*, using following command:

```
$ git branch fourthfile-branch
```

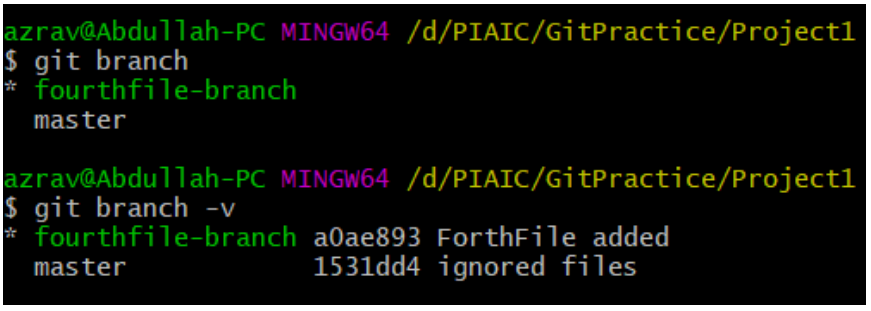
Display current branches

`git branch`

This command dispays all braches. Active or Checked out Branch is green in color.

`git branch -v`

This command shows some details. Active branch is colore in green. We can see first 7 digit of last commit hash and last commit message infront of each branch.



Switching to a branch

`git checkout <Branch>`

This command moves *head* from current branch to *Branch*. For example:

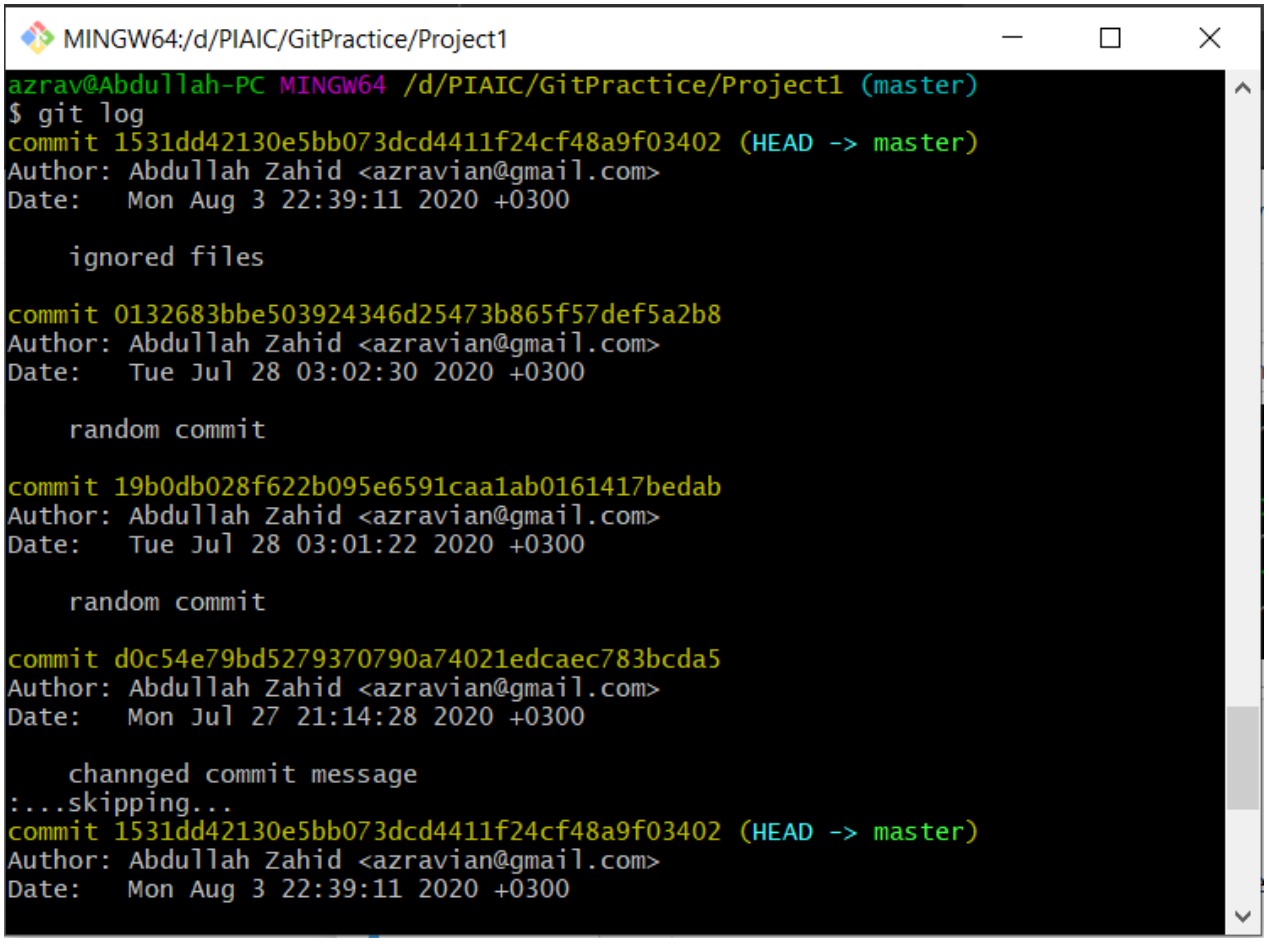
```
git checkout fouthfile-branch
```

command will switch head to *fourthfile-branch*.

1. Working Directory Contents depends on active branch: When working with branches, the working directory shows the contents exactly for that particular branch. For example, if we are on master branch we won't see *ForthFile.txt* in our working directory because this file was created when we were on *forthfile-branch*.
2. `git log` will display only commit made on that branch. When we make a commit in one branch it doesn't show in the logs of other branch. For example, I commited the addition of *ForthFile.txt* while on branch *forthfile-branch* `git log` shows following logs.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (forthfile-branch)
$ git log
commit d50915b36e19d75afbc03268f24365ff74852ab8 (HEAD -> forthfile-branch)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Tue Aug 4 00:33:52 2020 +0300

    Forth file added

commit 1531dd42130e5bb073dcd4411f24cf48a9f03402 (master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Mon Aug 3 22:39:11 2020 +0300

    ignored files

commit 0132683bbe503924346d25473b865f57def5a2b8
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Tue Jul 28 03:02:30 2020 +0300

    random commit

commit 19b0db028f622b095e6591caa1ab0161417bedab
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Tue Jul 28 03:01:22 2020 +0300

    random commit

commit d0c54e79bd5279370790a74021edcaec783bcda5
Author: Abdullah Zahid <azravian@gmail.com>
```

However, if I am on master branch and check logs, I won't get last commit with message: *"Forth File created"*



Merging Two Branches

`git merge <branch name>`

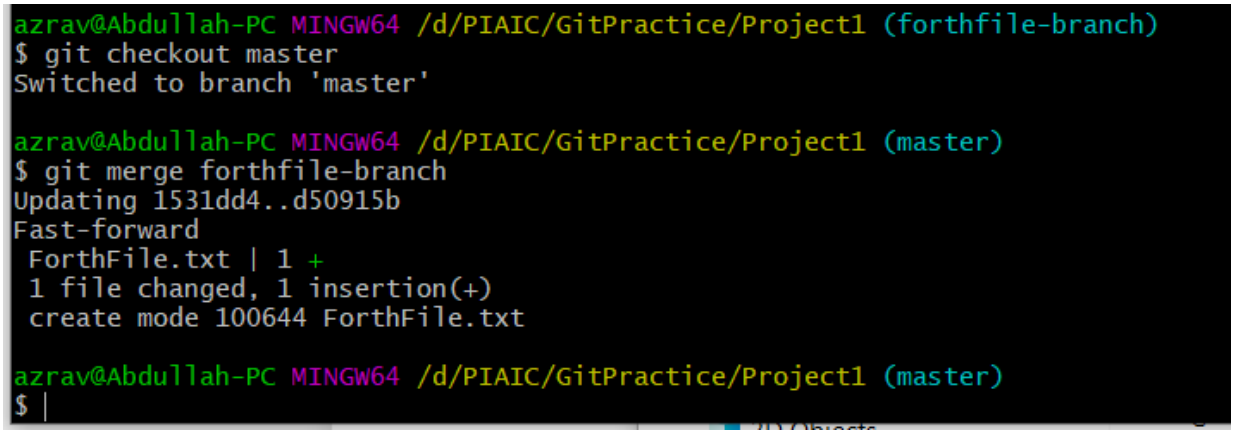
When all required chnages and modifications done for a feature/context, we would need to merge this feature branch with other branch.

When a branch is merged, all commits of this branch are merged with the branch it is merged with. Branching is done in two steps:

1. Change to branch we want to merged to.
2. Merge the required required.

for example, we had a branch *forthfile-branch* which we want to merge with *master* branch. Now first we need to *check-out* to *master* and then *merge forthfile-branch*.

```
$ git checkout master
$ git merge forthfile-branch
```



Now `git log` on master branch will also include the commit made on *forthfile-branch* as they are merge into *master* branch. The merging in only in one way, i.e. in the above examples, the commits from *ForthFile-branch* have been incorporated into *master* branch, but the commits of *master* branch are not added to *ForthFile-branch*

Difference of commits in two branches

`git log <Branch1>..<Branch2>`

This command would only show the commit that are in *Branch1* but not in *Branch2*

Delete a Loacal Branch

`git branch -d <BranchName>`

This command deletes a local branch. If we need to delete *forthfile-brach*:

```
$ git branch -d forthfile-branch
```

Saving Temporary Changes.

There might be a need when we have made some changes but are not complete for a commit and we need to switch to a branch. We can't do this unless we commit our changes, but commiting incomplete changes is not a good idea. In such case, we can save changes temporarily. It is done by using git's stash feature by using following command:

```
$ git stash
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
nothing to commit, working tree clean

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ForthFile.txt
        modified:   SecondFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash
Saved working directory and index state WIP on master: d50915b Forth file added

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Save stash with Message

```
git stash -m <message>
```

This command saves the stash with give message.

Save stash with a name

```
git stash save <StashName>
```

This command saves the stash with given name.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (RemoteBranch)
$ git stash -m "FirstFile"
Saved working directory and index state On RemoteBranch: FirstFile

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (RemoteBranch)
$ git stash list
stash@{0}: On RemoteBranch: FirstFile

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (RemoteBranch)
$ git stash save ForthFile
Saved working directory and index state On RemoteBranch: ForthFile

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (RemoteBranch)
$ git stash list
stash@{0}: On RemoteBranch: ForthFile
stash@{1}: On RemoteBranch: FirstFile
```

Listing all stashes:

```
git stash list
```

This command shows all stashes in git.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash list
stash@{0}: WIP on master: d50915b Forth file added
stash@{1}: WIP on master: d50915b Forth file added
stash@{2}: WIP on master: d50915b Forth file added
stash@{3}: WIP on master: d50915b Forth file added
```

The newest stash has the lowest number, and the older stashes have the higher number.

Restoring temporary changes

`git stash pop`

This command restore the temporary changes and clear them from the stored stashes.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash list
stash@{0}: WIP on master: d50915b Forth file added
stash@{1}: WIP on master: d50915b Forth file added
stash@{2}: WIP on master: d50915b Forth file added
stash@{3}: WIP on master: d50915b Forth file added

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   SecondFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (20fdaaac4f34f8caad218cd105bda44cbcf95e0c)

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   SecondFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash list
stash@{0}: WIP on master: d50915b Forth file added
stash@{1}: WIP on master: d50915b Forth file added
stash@{2}: WIP on master: d50915b Forth file added

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
```

Restoring a specific stash

`git stash apply <stash number>`

This command restore a specific stash specified by number. This command doesn't delete stash and have to separately deleted.

```
$ git stash apply 2
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   FirstFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

On may also use:

```
$ git stash apply stash@{2}
```

both above command and command in SS are same

Delete a stash

`git stash drop <StashNumber>`

This command deletes a stash specified bu the stash number.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git stash drop 2
Dropped refs/stash@{2} (03ebdf0bc39b01e781617a2f5200bd96124dfec6)
```

Delete All Stashes

`git stash clear`

This command clear all stashes.

Remote Repository.

Unitl now we worked with local repositories. Now we look into remote repositories. When working with remote repositories, basic flow is same. however after every commit we need to upload changes to remote repository, it's call *pushing* to remote and getting data from remote is called *pulling* data from remote. The basic workflow of git with remote repositories is:

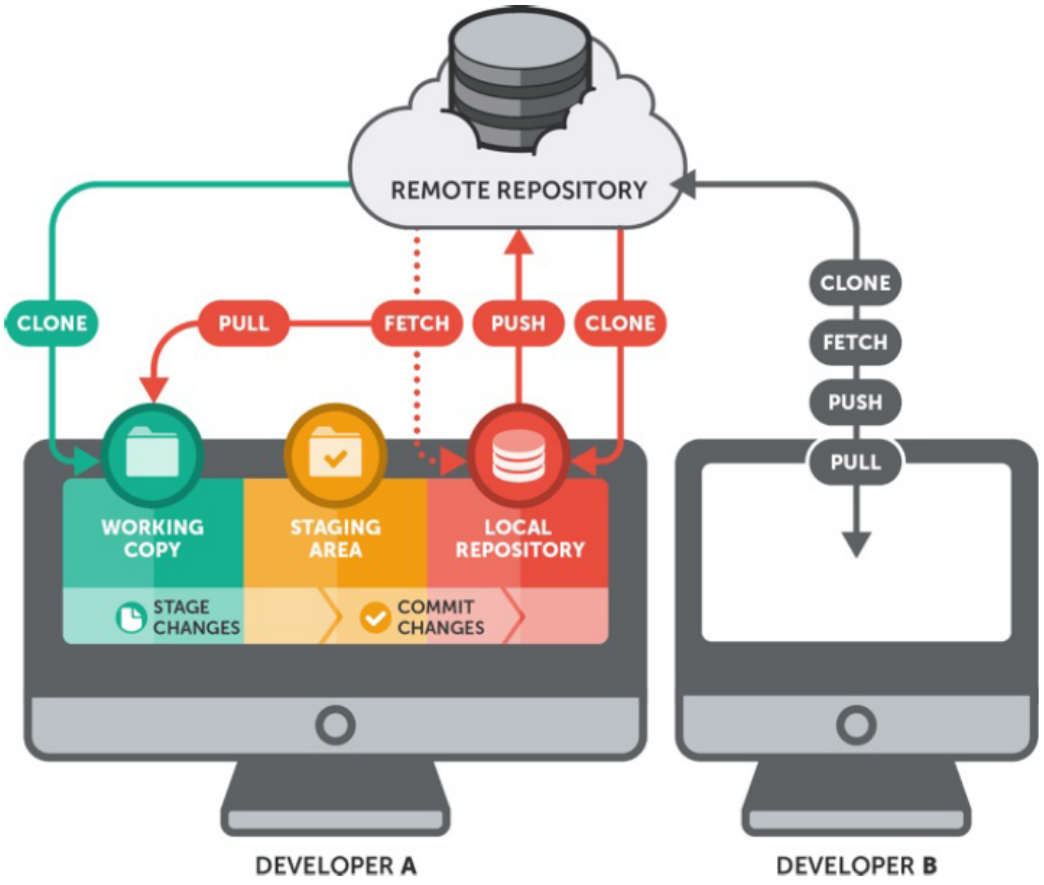


image ref: Learn Version control with Git

Remote Repository with multiple contributer.

while working with remote repository we need to take care of few things. For example, suppose 4 people are working on a project. At any instace, Person Push his changes to the remote repository, Now none of the rest of 3 people can make any push to remote unless they pull these changes from remote. It eliminates any risk of conflict.

Multiple people can work on a single file. If two people however, change code of same line of same file, A conflict is generated and we'll see this in details later.

Adding a remote repository

`git remote add <shortname> <remote address>`

For now I will use `https://github.com/azravian/PracticeProject.git` and I would name it as `origin` . Now my command for adding this repository will be:

```
$ git remote add origin https://github.com/azravian/PracticeProject.git
```

See List of all remote

`git remote -v`

This command displays all configured remote repositories.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git remote -v
origin  https://github.com/azravian/PracticeProject.git (fetch)
origin  https://github.com/azravian/PracticeProject.git (push)

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

Having multiple remote repositories.

In case we need to add more than one remote, we again have to use `git remote add` command followed by remote address:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git remote add second-repo https://github.com/azravian/Practice2.git

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git remote -v
origin  https://github.com/azravian/PracticeProject.git (fetch)
origin  https://github.com/azravian/PracticeProject.git (push)
second-repo     https://github.com/azravian/Practice2.git (fetch)
second-repo     https://github.com/azravian/Practice2.git (push)

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

See All branches and changes with remote

`git branch -va`

Now if we run this command all branches, local and remote are displayed.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git branch -va
  forthfile-branch            d50915b Forth file added
* master                      5eff23a Random commit
  remotes/origin/RemoteBranch 1b719d0 Merge pull request #1 from azravian/second-branh
  remotes/origin/master       1b719d0 Merge pull request #1 from azravian/second-branh
```

Get metadata from remote.

`git fetch <remote>`

This command gets the meta data from remote, however it doesn't incorporate the changes into the *Head* it means no file is changed after this command is executed. For example, we want to get metadata from *second-repo*:

```
$ git fetch second-repo
```

Create a local branch based on remote branch

`git checkout --track <remote/branch>`

In case we need to work on a remote branch, we need to create a branch based on this remote. for example is we want to work on *RemoteBranch* branch of remote *origin*. we use the following command:

```
$ git checkout --track origin/RemoteBranch
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git checkout --track origin/RemoteBranch
Switched to a new branch 'RemoteBranch'
Branch 'RemoteBranch' set up to track remote branch 'RemoteBranch' from 'origin'.
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (RemoteBranch)
$ git branch -va
* RemoteBranch                 1b719d0 Merge pull request #1 from azravian/second-branh
  forthfile-branch             d50915b Forth file added
  master                       5eff23a Random commit
  remotes/origin/RemoteBranch  1b719d0 Merge pull request #1 from azravian/second-branh
  remotes/origin/master        1b719d0 Merge pull request #1 from azravian/second-branh
```

Now a local branch have been created from a remote branch and this branch have been made active.

This branch is called *tracking branch* and tracks the remotes branch.
If local branch have some commits that are not published it is said to be *ahead* of remote branch.
if remote branch have more commits than that of local branch, local branch said to be *behind* the remote branch.

Suppose I create two commit on branch and before publishing these changes to remote repository, `git status` would tell the branch *ahead* or *behind* it remote counterpart.

```
$ git status
On branch RemoteBranch
Your branch is ahead of 'origin/RemoteBranch' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

above screenshot shows that local branch is ahead of remote branch by two commits.

Publishing all change to remote.

`git push <RemoteName> <BranchName>`

This command publish the changes on remote repositoy specified by RemoteName on branch specified by BranchName. For example, we want to publish all our changes to *second-repo* on *master* branch

```
$ git push second-repo master
```

However when we create a local counterpart branch of remote branch, when `git push` is executed without any arguments, git pushes all changes to the same branch of corresponding repository.

for example, if we are on *RemoteBranch* and execute following command:

```
$ git push
```

it will push all changes to *origin* on *RemoteBranch*.

Downloaind Data and incorporate into *Head*

`git pull <RemoteName>`

This command gets all data from remote repository and incorporate these changes into head of our local repository. For example, for my *origin* I need to run following command:

```
$ git pull origin
```

If we need to pull from only specific branch, we can also specify it:

```
$ git pull origin master
```

Publish a local branch to remote.

when we have created a local branch, suppose by name *LocalBranch*, Once we have made a commit to this branch we need to push this local branch to remote repository *origin*. We use following command

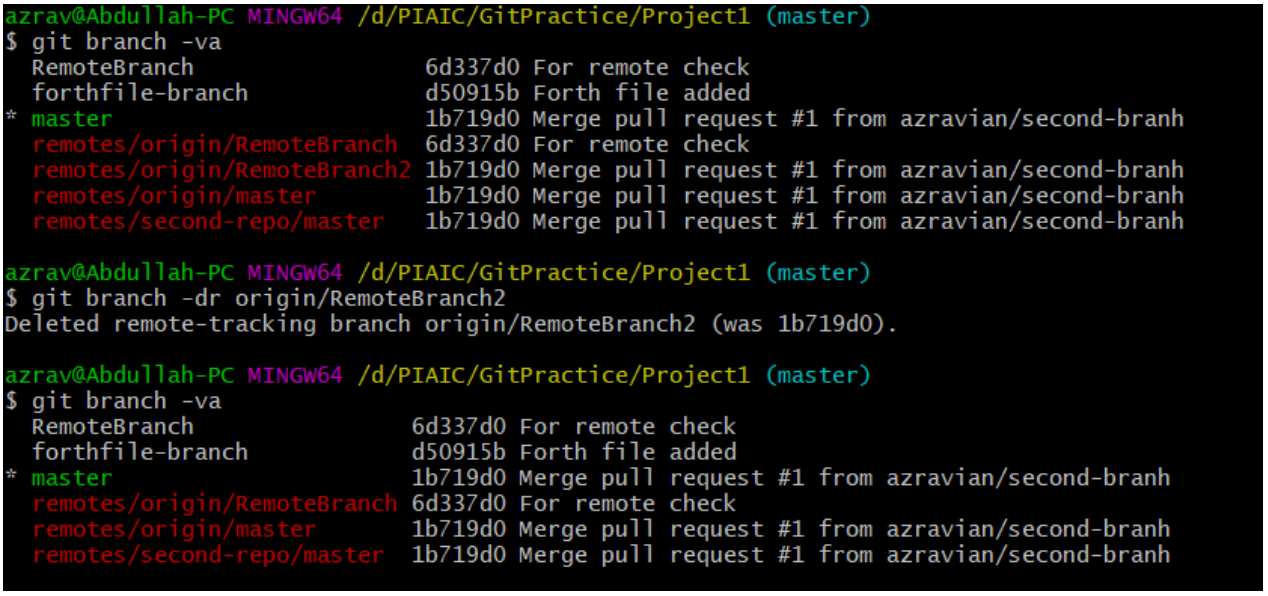
```
$ git push origin LocalBranch
```

This command not only pushes the commit to *origin* but also the branch *LocalBranch*

Delete a tracking remote branch

`git branch -dr <RemoteName>/<BranchName>`

This command deletes the local tracking copy of remote branch. However the branch on the remote won't be deleted.

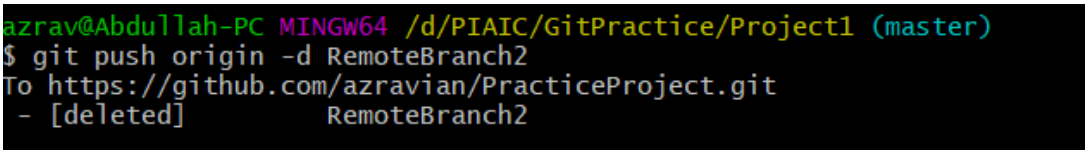


Deleting a branch on remote

`git push <RemoteName> -d <BranchName>`

This command delete a branch on remote. For example, I wanna delete RemoteBranch2 on origin.

```
$ push origin -d RemoteBranch2
```



Remove a remote from git

`git remote rm <RemoteName>`

This command removes the specified remote from local repository, let us remove second-repo from our repository using following command:

```
$ git remote rm second-repo
```

Advanced topics

Addition to most recent commit.

`git add`

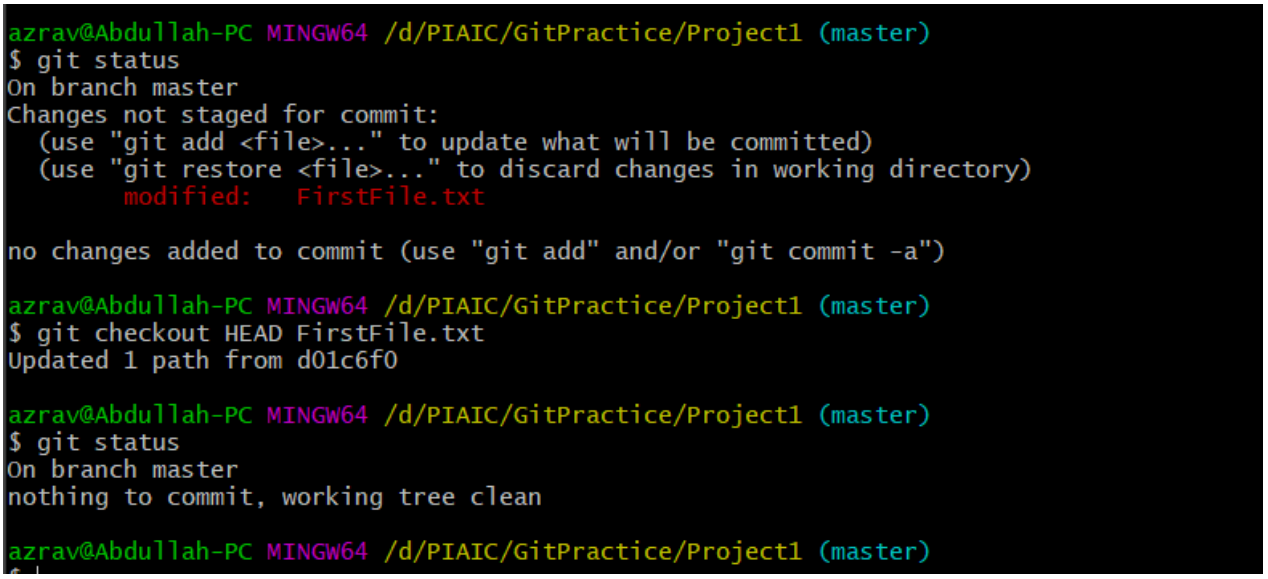
`git commit --amend -m "Message"`

previously we have seen that this command ammends the message of most recent commit. But in case we have staged changes we can incorporate those changes into the most recent commit.

Undoing Uncommitted Changes

`git checkout HEAD <filename>`

This command restores the file to last commit. This command only restore uncommited changes in the specified file. These changes can't be undone.



`git reset --hard HEAD`

This command retore all uncommited changes in all files to the last commit. Once reset is done, it cannot be undone.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   FirstFile.txt
        modified:   SecondFile.txt
        modified:   ThirdFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git reset --hard HEAD
HEAD is now at 8137e5c First file deletion

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Undoing Committed change

`git revert <commit hash>`

This commit reverts the working directory to specified commit. Only first seven characters of commit hash are enough to specify a commit.

This command does not delete any commit, but get the changes from sprcified commit and create a new commit with those changes. Even after reverting to previous commit, all commit will be visible in logs.

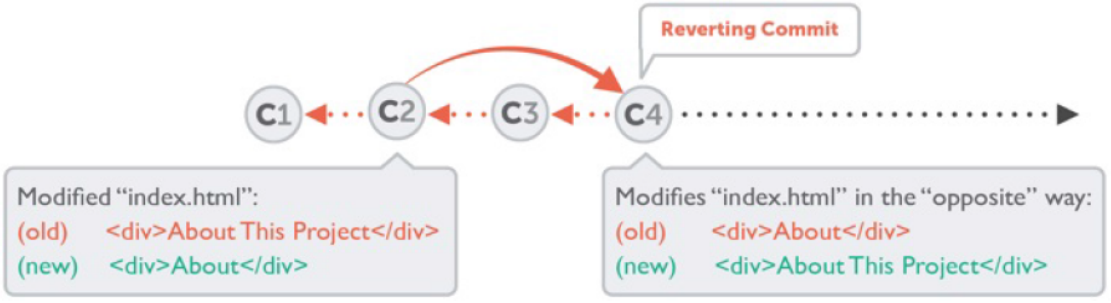


image ref: Learn Version control with Git

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git revert 64fea57
[master 5089214] Revert "Update SecondFile.txt"
 1 file changed, 5 insertions(+), 1 deletion(-)
```

`git reset --hard <commit hash>`

This command moves the *HEAD* to a commit specified by commit hash. This command won't add ant commit nor will delete any commit. But as our *HEAD* now moved to previous commit, the commit after this commit won't be visible in our logs. For example, we have commit represented as C1, C2, C3, C4 and currently head is on C4. If we *reset hard* to C2, all file will have only change made in this commit. Now *Head* will be at C2, and C3 and C4 won't show in logs. Since all these commits are not deleted, they are only hidden from history and stays in the memory for the *30 days*.

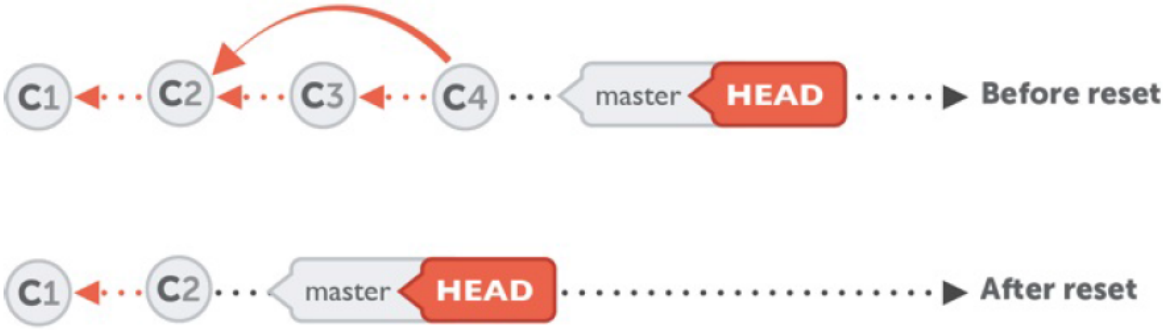


image ref: Learn version control with git

```
git reset --keep <commit hash>
```

This command work same as above command, except it will keep all changes from the between most recent commit and commit which we are resetting to. In above example, if *--keep* is used instead of *--hard* all changes from commit *C2 and C3* will also be kept in the files.

Recovering a reset commit.

As we have seen that *rest --hard--* move head to specified commit and all intermediate commits are now not visible in *logs* and they are neither deleted, but are kept for *30 days*. If we need to recover such commit, we need to two steps first run the following command:

```
$ git reflog
```

and note down the commit hash we want to recover. and run following command. ``$ git reset

Showing above *reset*

Resetting to previous commit

Current commit history.

Current my *git log* have these recent commits:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log --pretty=oneline
f8f9124be43bfd5a52fabcc74308006263496386 (HEAD -> master) First File updat
346c15dae1818f85cb8d231f03965adbec0c9366 Third File updated
53a2707c0685c2c086274d50b2f1657b935755b5 Second File updated
4f6b918ea231c9de4b5e1589f8551fa7e467e0aa Forth File updated
5e8a900bda958b83f8965b44b5325779e0a22af6 First File Modified
5089214e008784e9d2e591e9e8f50c38ccbae708 Revert "Update SecondFile.txt"
8137e5cacd439517b5de63b2685b42efa0333d9c First file deletion
3c9d9039ea96c39c0dd0ce9afdd0b5e8d3d05b99 FirstFile.txt updated
```

reset to previous commit

let us reset our repository to a commit identified by commit hash: *5e8a900bda958b83f8965b44b5325779e0a22af6* sicne only first seven characters are necessary. we need to use following command:

```
$ git reset --hard 5e8a900
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git reset --hard 5e8a900
HEAD is now at 5e8a900 First File Modified
```

Now our head has moved back. Now checking logs again:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log --pretty=oneline
5e8a900bda958b83f8965b44b5325779e0a22af6 (HEAD -> master) First File Modified
5089214e008784e9d2e591e9e8f50c38ccbae708 Revert "Update SecondFile.txt"
8137e5cacd439517b5de63b2685b42efa0333d9c First file deletion
3c9d9039ea96c39c0dd0ce9afdd0b5e8d3d05b99 FirstFile.txt updated
1b719d0bb58bca5cf5c1798c5c6dafa2d92bc192 (second-repo/master, origin/master) Me
```

Notice all commits after the commit we have reset to are no more visible in log.

Recovering the commit.

Find the commit hash which is to be restored.

enter following command

```
$ git reflog
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git reflog
5e8a900 (HEAD -> master) HEAD@{0}: reset: moving to 5e8a900
f8f9124 HEAD@{1}: commit: First File updated
346c15d HEAD@{2}: commit: Third File updated
53a2707 HEAD@{3}: commit: Second File updated
4f6b918 HEAD@{4}: commit: Forth File updated
5e8a900 (HEAD -> master) HEAD@{5}: commit: First File Modified
5089214 HEAD@{6}: revert: Revert "Update SecondFile.txt"
8137e5c HEAD@{7}: reset: moving to HEAD
```

Restore the required commit

notice that it has the position of head over the time. For now we need to restore the commit identified by *53a2707*. we need following command:

```
$ git reset 53a2707
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git reset 53a2707
Unstaged changes after reset:
M       ForthFile.txt
M       SecondFile.txt
```

Now our files contains the changes from this commit and now we need to stage and commit these changes.

Push after resetting.

When we have reset out repository to some previous commit, the local branch lags the remote branch. If we try to push in this condition, the git generates an error.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git push origin master
To https://github.com/azravian/PracticeProject.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/azravian/PracticeProject.
git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

At this stage, the git asks us to pull changes from remote, but if we do so, the the changes we have reset will again be incorporated in our local repository. So in this case we need to *force* push our reset to our remote by using following command:

```
$ git push -f orignin master
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git push -f origin master
Enumerating objects: 73, done.
Counting objects: 100% (73/73), done.
Delta compression using up to 8 threads
Compressing objects: 100% (53/53), done.
Writing objects: 100% (65/65), 9.88 KiB | 919.00 KiB/s, done.
Total 65 (delta 21), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (21/21), completed with 3 local objects.
To https://github.com/azravian/PracticeProject.git
 + 9c1938a...bec318b master -> master (forced update)
```

Detailed Changes with `git diff`

`git diff` command shows the uncommited changes in the current working files and last commit. for example we made some changes to a file.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git diff
diff --git a/FirstFile.txt b/FirstFile.txt
index 46a05b8..6ce666a 100644
--- a/FirstFile.txt
+++ b/FirstFile.txt
@@ -3,6 +3,6 @@ added1
 added2
 added3
 added4
-added5
-added6
-added7
+Replaced
+three
+lines
\ No newline at end of file
diff --git a/SecondFile.txt b/SecondFile.txt
index 8b13789..957303b 100644
--- a/SecondFile.txt
+++ b/SecondFile.txt
@@ -1 +1 @@
-
+Addition to the file
```

it shows all the changes made, the details of each segment is as below:

Compared files a/b:

The fist line shows the files being compared. These are same file with different versions. In this example, it is as follow:

```
diff --git a/FirstFile.txt b/FirstFile.txt
```

as it can be seen that two versions of *FirstFile.txt* are being compared.

Metdata:

The second line shows the metadata. which in our case is:

```
index 46a05b8..6ce666a 100644
```

the first two numbers are hashes for each version for this file, *46a05b8* for version *a* and *6ce666a* for version *b*. The next number is file mode identifier. 100644 is used for "normal file", 100755 for an executeable and 120000 for a symbolic link

Markers:

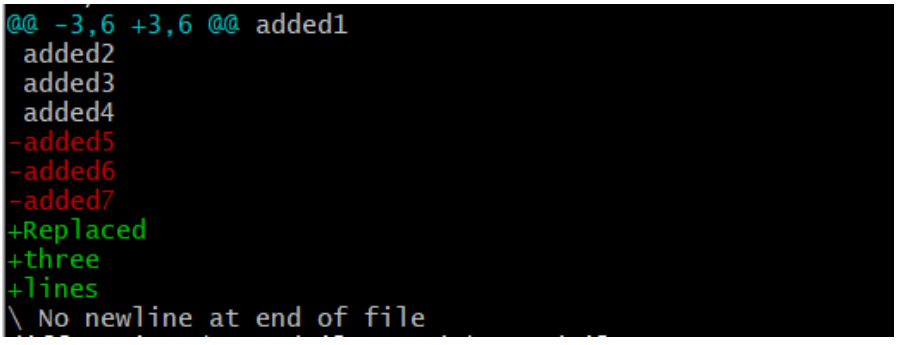
Next comes the markers for a and b versions of file. in the case above the markers are:

```
--- a/FirstFile.txt
+++ b/FirstFile.txt
```

It shows that `-` is being for version *a* of our file, and `+` is being used for the version *b* of the file.

Chunk(Hunk):

Git doesn't show the content of whole files, but it only show the portion of file which have been modified. These are called chunk(or hunk). A chunk does also include some unchanged lines in order to make it more understandable. In our example there are two chunks, one of them are:



Chunk Header:

The first line in a chunk is header which is enclosed in two @, which in our example is:

```
@@ -3,6 +3,6 @@ added1
```

as we have discussed previously that `-` is for *version a* and `+` *version b* of the file.
Hence **-3, 6** shows theinformation about the displayed lines from *version a* and it means that 6 lines are being shown starting from line 3.
Similary, **+3, 6** shows the information about the lines displayed from *version b* and it means that 6 line are displayed starting from line 3

After that we have some lines with white text, these are the unchanged line in both versions of file.
then comes the lines starting with a `-` and with font color which are from *verion a* and are removed.
the next part of chunk have lines starting with `+` with font color of green and are show the content from *version b*

commits details

When `git log` is used, it shows the basic details, such as commit hash, author name and timestamp. if any time we wish to show more than just commit hash, author name we can use following command

```
git log -p
```
or
```
git log --patch
```

It will display all the changes made in all commits.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log -p
commit 3dff7a1a4abb1497b5842d11726483af29e02b6a (HEAD -> master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 13:45:27 2020 +0300

    Recent modifications

diff --git a/FirstFile.txt b/FirstFile.txt
index 46a05b8..6ce666a 100644
--- a/FirstFile.txt
+++ b/FirstFile.txt
@@ -3,6 +3,6 @@ added1
 added2
 added3
 added4
-added5
-added6
-added7
+Replaced
+three
+lines
\ No newline at end of file
diff --git a/SecondFile.txt b/SecondFile.txt
index 8b13789..957303b 100644
--- a/SecondFile.txt
+++ b/SecondFile.txt
@@ -1 +1 @@
-
+Addition to the file

commit f75dcdc097777cd56d129a9f2481f9ff4044520c (origin/master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 11:45:42 2020 +0300

    addition of files.

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..7016e76
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,21 @@
+# uncomment line below for Mac OS system
+#.DS_Store
+
+# ignore html file.
+
+*.html
+
+#ignore python files
+
+*.py
+
+# ignore src/index.txt
+src/index.txt
+
+# ignore data folder
```

Single commit details

The command above shows details of all commits. In case we need the details of single commit we use following command.

```
git log -p -1 <commit hash>
```

for example, we need to see the details of latest commit identified by a hash of *3dff7a1*,

```
git log -p -1 3dff7a1
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log -p1 3dff7a1
fatal: unrecognized argument: -p1

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log -p -1 3dff7a1
commit 3dff7a1a4abb1497b5842d11726483af29e02b6a (HEAD -> master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 13:45:27 2020 +0300

    Recent modifications

diff --git a/FirstFile.txt b/FirstFile.txt
index 46a05b8..6ce666a 100644
--- a/FirstFile.txt
+++ b/FirstFile.txt
@@ -3,6 +3,6 @@ added1
 added2
 added3
 added4
-added5
-added6
-added7
+Replaced
+three
+lines
\ No newline at end of file
diff --git a/SecondFile.txt b/SecondFile.txt
index 8b13789..957303b 100644
--- a/SecondFile.txt
+++ b/SecondFile.txt
@@ -1 +1 @@
-
+Addition to the file

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
```

Comparing two branches

In case we need to compare two branches, we need to use following command:

```
$ git diff branch1..branch2
```

for example, we have two branch `master` and `dev`, when we need to compare these two brnaches

```
$ git diff master..dev
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git diff master..dev
diff --git a/FirstFile.txt b/FirstFile.txt
index 92b9278..6ce666a 100644
--- a/FirstFile.txt
+++ b/FirstFile.txt
@@ -5,6 +5,4 @@ added3
 added4
 Replaced
 three
-lines
-
-a change on master
\ No newline at end of file
+lines
\ No newline at end of file
```

Comparing two versions.

We can comapring two version using the following command.

```
$ git diff <hash1>..<hash2>
```

hash1 defines *the version a* and *hash2* defines the *version b*. For example, for my repository I have two version specified by two hashes, 5af716c and bb879a2.

Now comparing these two versions:

```
$ git diff 5af716c..bb879a2
```

It is kept in mind that *5af716c* specifies *version a* and *bb879a2* specifies *version b*.

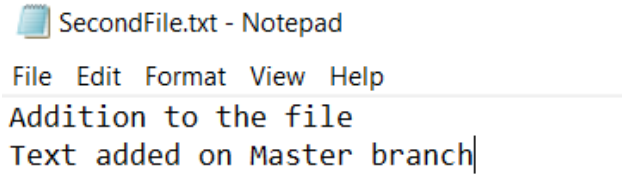
```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git diff 5af716c..bb879a2
diff --git a/FirstFile.txt b/FirstFile.txt
index 2378545..92b9278 100644
--- a/FirstFile.txt
+++ b/FirstFile.txt
@@ -7,5 +7,4 @@ Replaced
 three
 lines

-a change on master
-second change on master
\ No newline at end of file
+a change on master
\ No newline at end of file
```

Resolving Conflicts during merging.

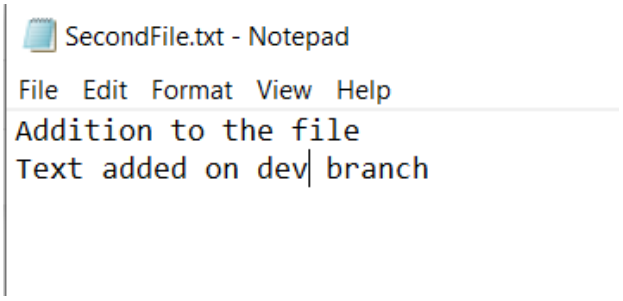
When we have to merge two branches, where we have changes on same lines, a conflict is generated and branches can not be merged.

For example, I added some text to my *SecondFile.txt* at second line: "Text added on Master branch" for simplicity and commit these changes. Then *checked out* to dev branch and added "Text added on dev branch" to same file on same line.

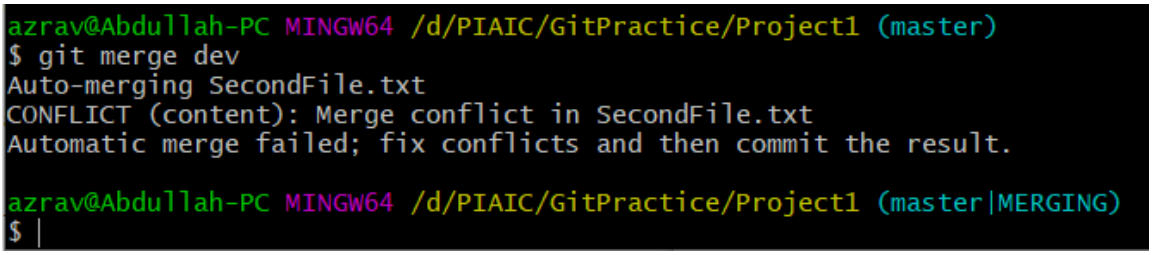
SecondFile.txt have following contents on *master* branch:



the same file have following contents on *dev* branch:



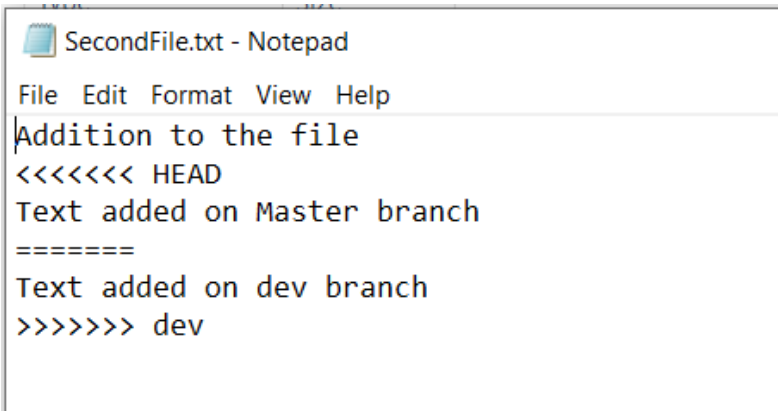
Now same file have different content on same lines. Now let us merge our *dev* into *master*



Now notice an conflict has appeared, and our git is in merging state. This merge can be solved using two methods:

Resolving conflicts using text Editor:

At this point open file in text editor, and we will notice that file content have been changed. It would like as below:



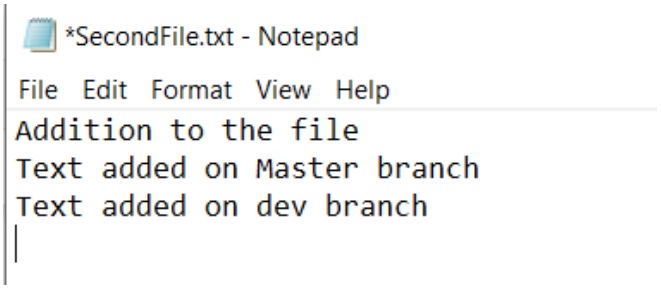
The first line is unchanged. and

```
<<<<<<< HEAD
Text added on Master branch
=======
```

Show the changes made on master branch, as our *HEAD* is currently on *master* branch.

```
=======
Text added on dev branch
>>>>>>> dev
```

shows the changes made in dev branch. Now we can edit our file and finalize the changes we need to keep after merge. For now I want to keep both change, So I changed my file to as follow:



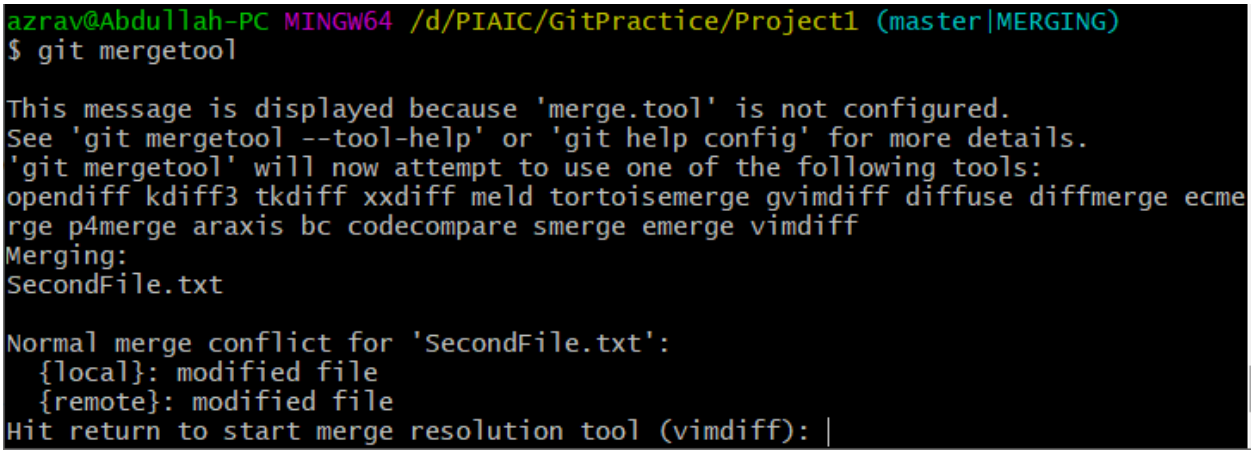
Now once conflict resolved, we need to commit this merge.

Resolving Commit `mergetool`

Another method to reolve the conflict is by using following command:

```
git mergetool
```

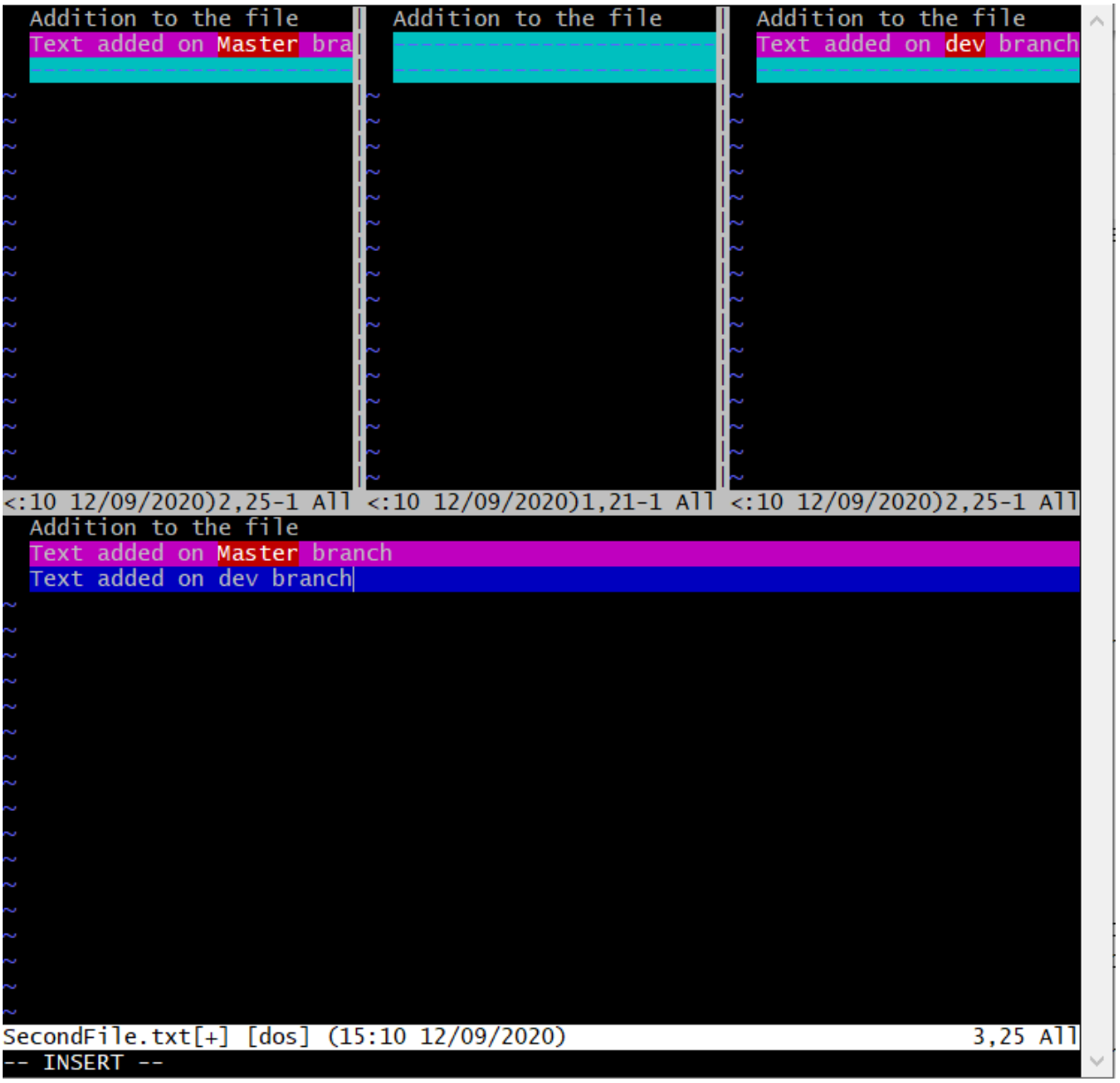
It will start a mergetool in git bash. For now I will use *vimdiff* for this purpose. once command is run the git prompts if we want to continue.



and just press enter and git will start *vimdiff mergetool*

The upper-left section shows the contents of current branch i.e *master*, and the upper-right section shows the content of other branch which is being merged that is *dev*. The upper-center section shows the final content. The bottom section of window is editor.

At this stage press `i` and we will enter into *insert more* and then we can make changes in the editor of vim.

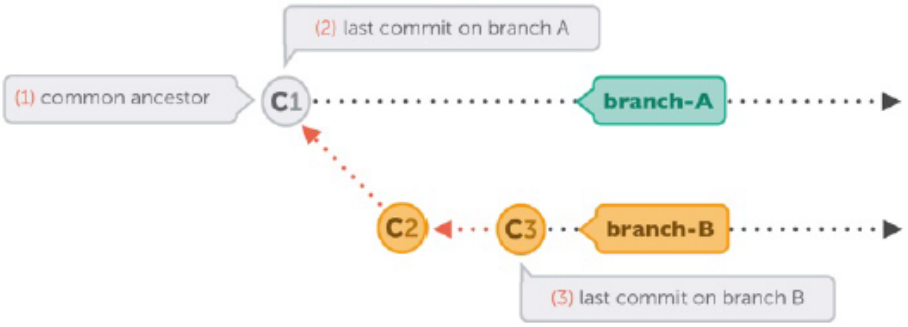


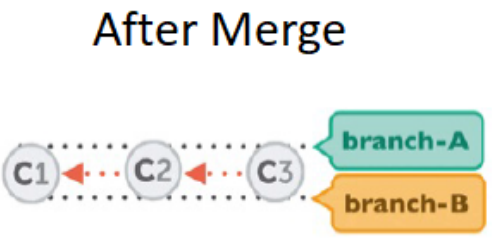
Once finalized, press `esc` and the `:wq` and hit `enter`. In all following windows, enter `:q` as long as git bash isn't out of mergetool. once out of mergetool, just commit the merge.

Merge Fast-Forward.

In case we have a situation when a one branch have some commits while the other branch have not commit. In other words, the latest commit on any branch is still the ancestor commit; the commit after which the branching is done. In this case merge is done simply by adding the commits from one branch to other branch after the ancestor commit.

For example, the Branch B is created after commit C1, and and then two commits were made on this branch while no commit was done on Branch A. The latest commit on Branch A is still C1. Now if we merge these branches, all commits from branch B will be added after C1 commit.





let us have an example with our files, first we create a branch *dev* from *master*. At this point our branches, *master*, *dev* and remote branch *origin/master* are at same commit which is identified by hash *dfa344d*.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit dfa344d35ccc59f542e7f23db39e601606e54572 (HEAD -> master, origin/master, dev)
Merge: 5e2ba48 fc44f6d
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 15:20:30 2020 +0300

    merge dev into master
```

at this point I will switch to *dev* branch and create two commit on this branch identified by *810b81c* and *fc70732*. Now dev branch is two commits ahead of master.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (dev)
$ git log
commit fc70732ac64d27e0c93c4a7a0cf0a42cd391527a (HEAD -> dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:15:26 2020 +0300

    For fast forward merge C3

commit 810b81c5e33e403e013eed642e9f34182c2f8c2b
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:15:11 2020 +0300

    For fast forward merge C2

commit dfa344d35ccc59f542e7f23db39e601606e54572 (origin/master, master)
Merge: 5e2ba48 fc44f6d
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 15:20:30 2020 +0300
```

Now let us merge *dev* into *master*. As we merge these branches, the resulting message shows that merge was done with fast-forward.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (dev)
$ git checkout master
Switched to branch 'master'

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git merge dev
Updating dfa344d..fc70732
Fast-forward
 FirstFile.txt  | 3 ++-
 SecondFile.txt | 1 +
 2 files changed, 3 insertions(+), 1 deletion(-)

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

Now if we check logs:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit fc70732ac64d27e0c93c4a7a0cf0a42cd391527a (HEAD -> master, dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:15:26 2020 +0300

    For fast forward merge C3

commit 810b81c5e33e403e013eed642e9f34182c2f8c2b
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:15:11 2020 +0300

    For fast forward merge C2

commit dfa344d35ccc59f542e7f23db39e601606e54572 (origin/master)
Merge: 5e2ba48 fc44f6d
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 15:20:30 2020 +0300
```

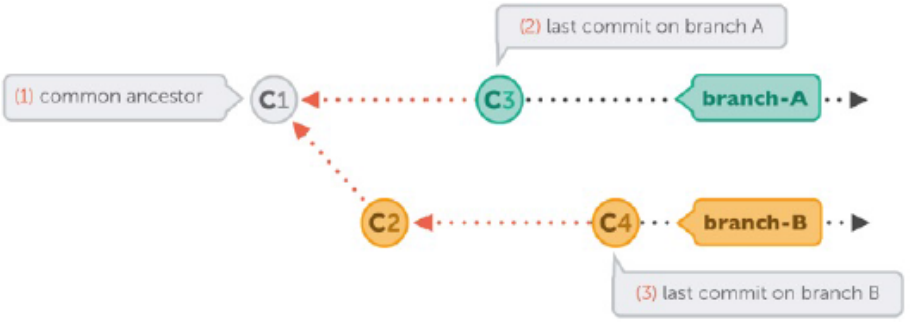
Notice that the last commit is still *fc70732* which was created on *dev* and merging have not created any new commit for or after this merge. we can also see this in git graph.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log --oneline --graph
* fc70732 (HEAD -> master, dev) For fast forward merge C3
* 810b81c For fast forward merge C2
*   dfa344d (origin/master) merge dev into master
```

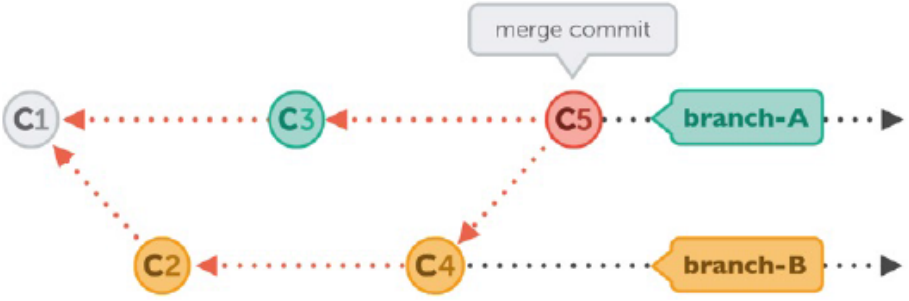
notice all three commits are aligned in a line and no new commit is generated.

Merge Commit.

Now if we have a case when both branches have some commits after the bracnhing is done, *fast-forward* merging cannot be done as both branches have moved individually. For example, a new Branch B is created from Branch A after commit C1, and then a commit is made on Branch B, and then a commit is made on Branch A, and then again on Branch B. Now both Branch A and Branch B have their own commit, in this case when doing merging, git witll create a commit automatically, so this type of merging is called *Merge-Commit*.



After Commit:



Now using our example, let's create a branch *dev* from *master*. Current both branch are at commit *2441db9*.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit 2441db95b66f5d4e58e58f6c506739babdff4a30 (HEAD -> master, dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:40:13 2020 +0300

    first file updated
```

Now let us create two commits on *dev* and one commit on *master*.

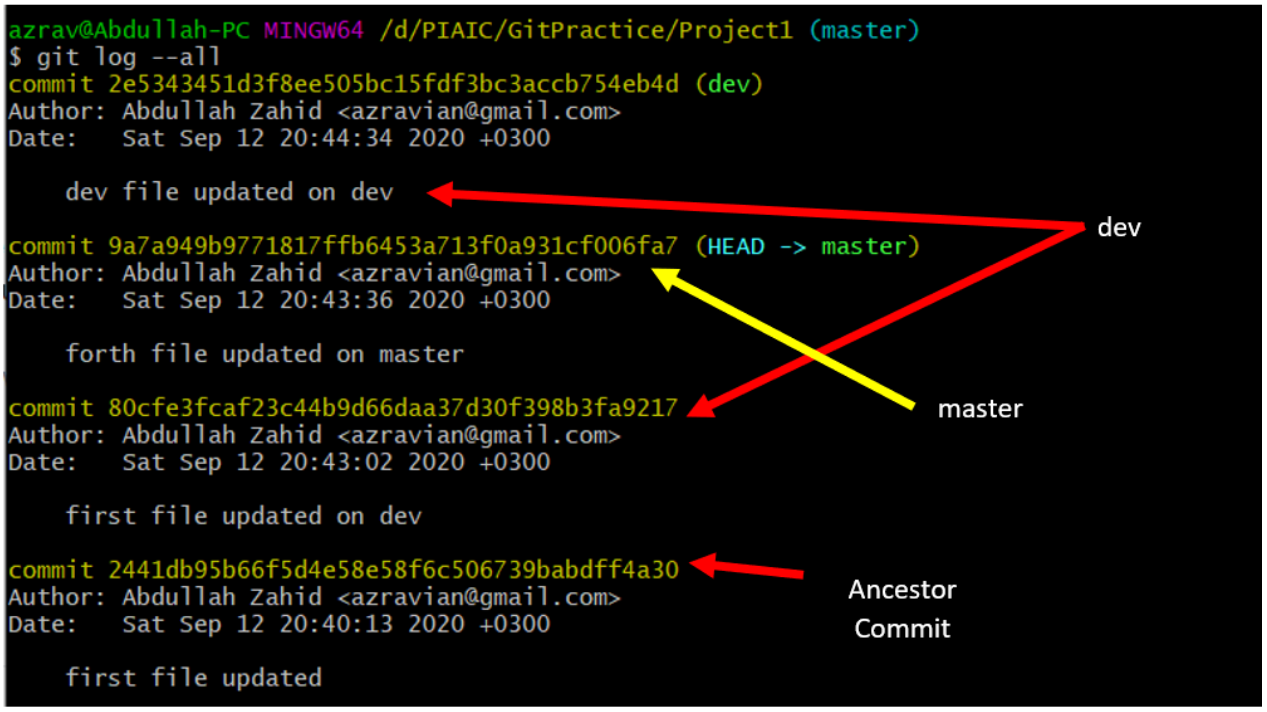
Logs on our *dev* branch:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (dev)
$ git log
commit 2e5343451d3f8ee505bc15fdf3bc3accb754eb4d (HEAD -> dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:44:34 2020 +0300

    dev file updated on dev

commit 80cfe3fcaf23c44b9d66daa37d30f398b3fa9217
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:43:02 2020 +0300

    first file updated on dev
```

Logs on master branch:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit 9a7a949b9771817ffb6453a713f0a931cf006fa7 (HEAD -> master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:43:36 2020 +0300

    forth file updated on master

commit 2441db95b66f5d4e58e58f6c506739babdff4a30
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:40:13 2020 +0300

    first file updated
```

and combined logs for ease



Now when *dev* is merged into master, we get following result.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git merge dev
Merge made by the 'recursive' strategy.
 FirstFile.txt  | 2 +-
 SecondFile.txt | 3 ++-
 2 files changed, 3 insertions(+), 2 deletions(-)
```

Notice "Merge made by the 'recursive' strategy." not "fast-forward". Now checking logs again:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit 40ea910cf06caeff2b11a3edabfc0df52a4fed48 (HEAD -> master)
Merge: 9a7a949 2e53434
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 21:01:01 2020 +0300

    Merge branch 'dev'

commit 2e5343451d3f8ee505bc15fdf3bc3accb754eb4d (dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:44:34 2020 +0300

    dev file updated on dev

commit 9a7a949b9771817ffb6453a713f0a931cf006fa7
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:43:36 2020 +0300

    forth file updated on master

commit 80cfe3fcaf23c44b9d66daa37d30f398b3fa9217
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:43:02 2020 +0300

    first file updated on dev

commit 2441db95b66f5d4e58e58f6c506739babdff4a30
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sat Sep 12 20:40:13 2020 +0300

    first file updated
```

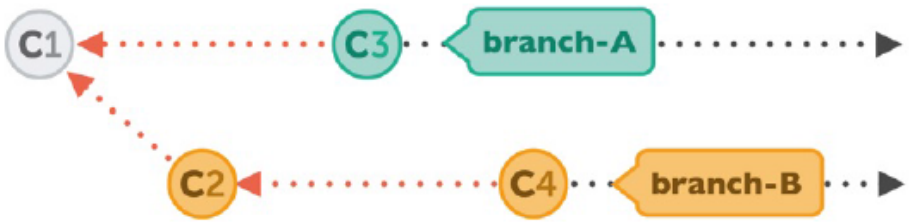
Now a new commit, with hash of *40ea910*, have been added. This is a merge commit and is generated automatically.

Rebase.

`git rebase <BranchName>`

Somethimes it is prefered not to have merge commit, rather it is desired to have a case in which looks like as if the project have evolved over a straight line and it seems that project was never split into multiple branches.

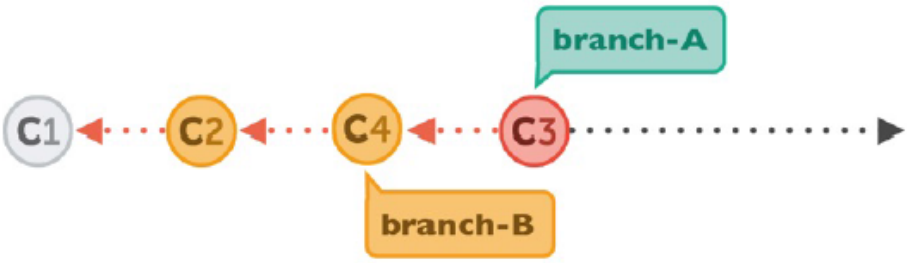
Let us focus on example, at any point a *Branch-B* from *Branch-A* after a commit *C1*, ancestor commit. Then a commit is made to Branch-B *C2*, and then on Branch-A *C3* and at then end a commit *C4* was made to Branch-B. The history looks like as follow:



Now at this stage, if we need to rebase our *Branch-B* to *Branch-A*, we use following command:

```
git rebase Branch-B
```

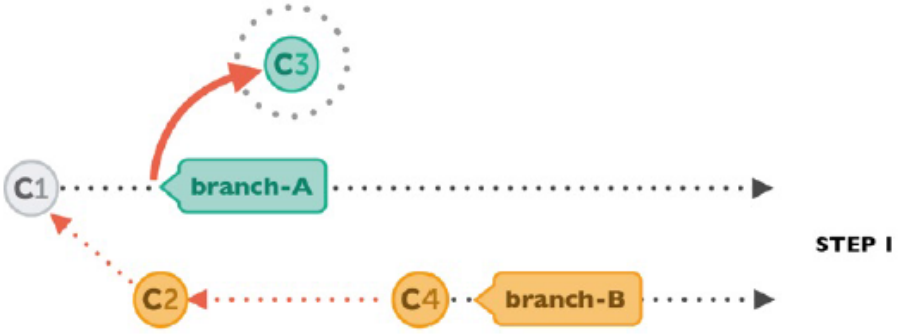
Now the history would be like following:



How Rebasing is done:

Step 1: Temporarily undoing commit on Branch A:

All the commits made on Branch-A, *C3* in this example, are undone temporarily and are saved away.

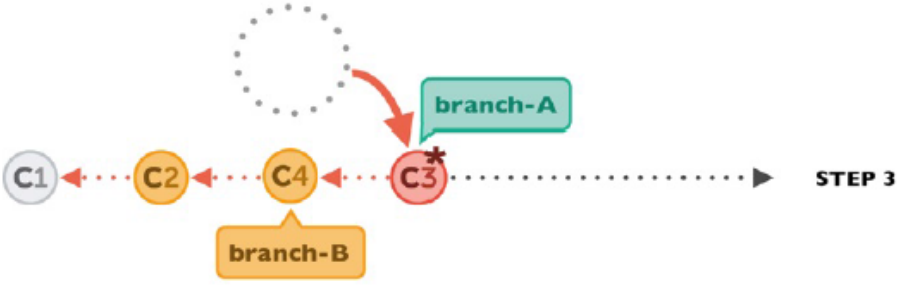


Step 2: Applying commits from Branch B to Branch A:

At this point the commits of Branch-B, *C2* and *C4* are applied to Branch-A. And at this point both branches are identical.

Step 3: Apply back the undone commit to Branch-A:

Now the The commit which had been temporarily udone and saved away are applied back to Branch-A.



Notice an asterick with *C3* as it is different from *C3* w.r.t. to actual history.

images ref: Learn Version Control with Git.

Example:

Now let's move to an example with git. Let us create a branch *dev* from *master*. The commit identified by hash *24a811b* is now *ancestor commit*. I have done following step now:

1. Commit some change on dev branch, commit Hash: *0d82558*: "FirstFile.txt modified on dev for rebase"
2. Commit some change on master branch, commit Hash: *ed77e45*: "SecondFile.txt modified on master for rebase"
3. Commit some change on dev branch, commit Hash: *0d82558*: "ForthFile.txt modified on dev for rebase"

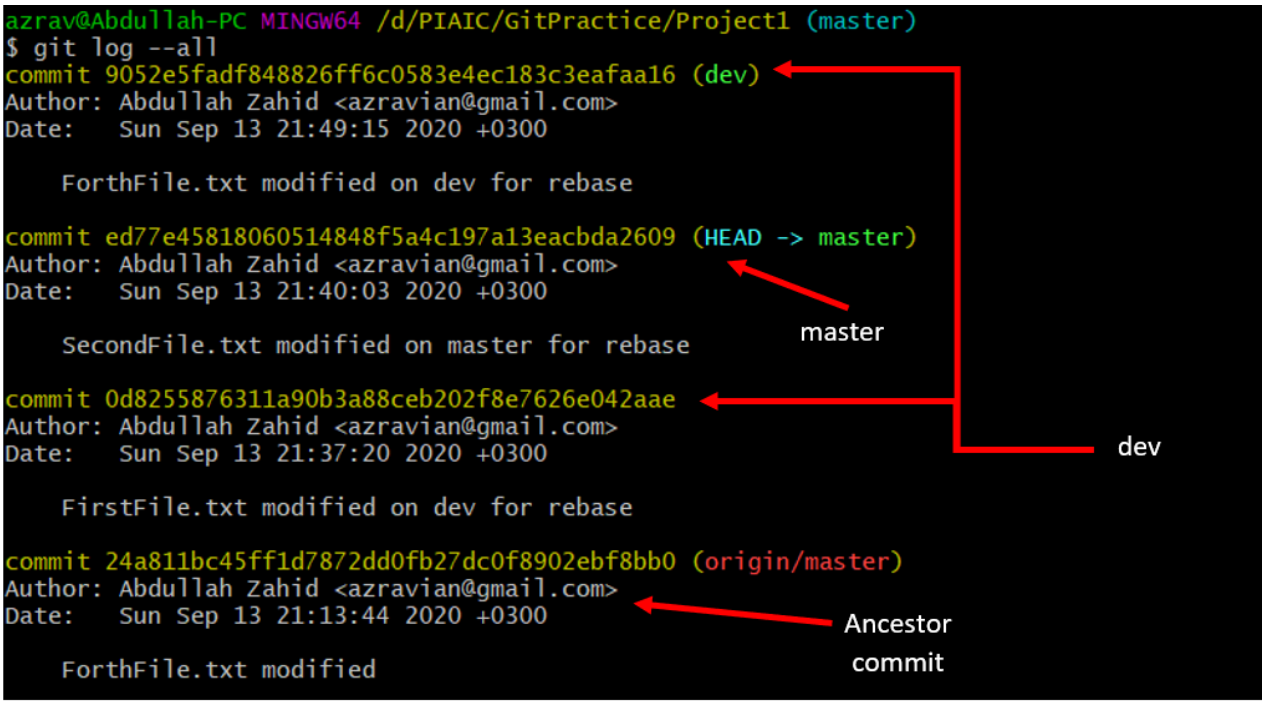
git log on dev branch:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (dev)
$ git log
commit 9052e5fadf848826ff6c0583e4ec183c3eafaa16 (HEAD -> dev)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sun Sep 13 21:49:15 2020 +0300

    ForthFile.txt modified on dev for rebase

commit 0d8255876311a90b3a88ceb202f8e7626e042aae
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sun Sep 13 21:37:20 2020 +0300

    FirstFile.txt modified on dev for rebase

commit 24a811bc45ff1d7872dd0fb27dc0f8902ebf8bb0 (origin/master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sun Sep 13 21:13:44 2020 +0300

    ForthFile.txt modified
```

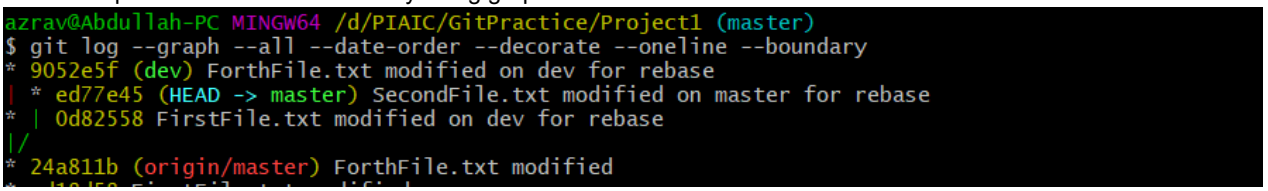
git logs on master branch:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git log
commit ed77e45818060514848f5a4c197a13eacbda2609 (HEAD -> master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sun Sep 13 21:40:03 2020 +0300

    SecondFile.txt modified on master for rebase

commit 24a811bc45ff1d7872dd0fb27dc0f8902ebf8bb0 (origin/master)
Author: Abdullah Zahid <azravian@gmail.com>
Date:   Sun Sep 13 21:13:44 2020 +0300

    ForthFile.txt modified
```

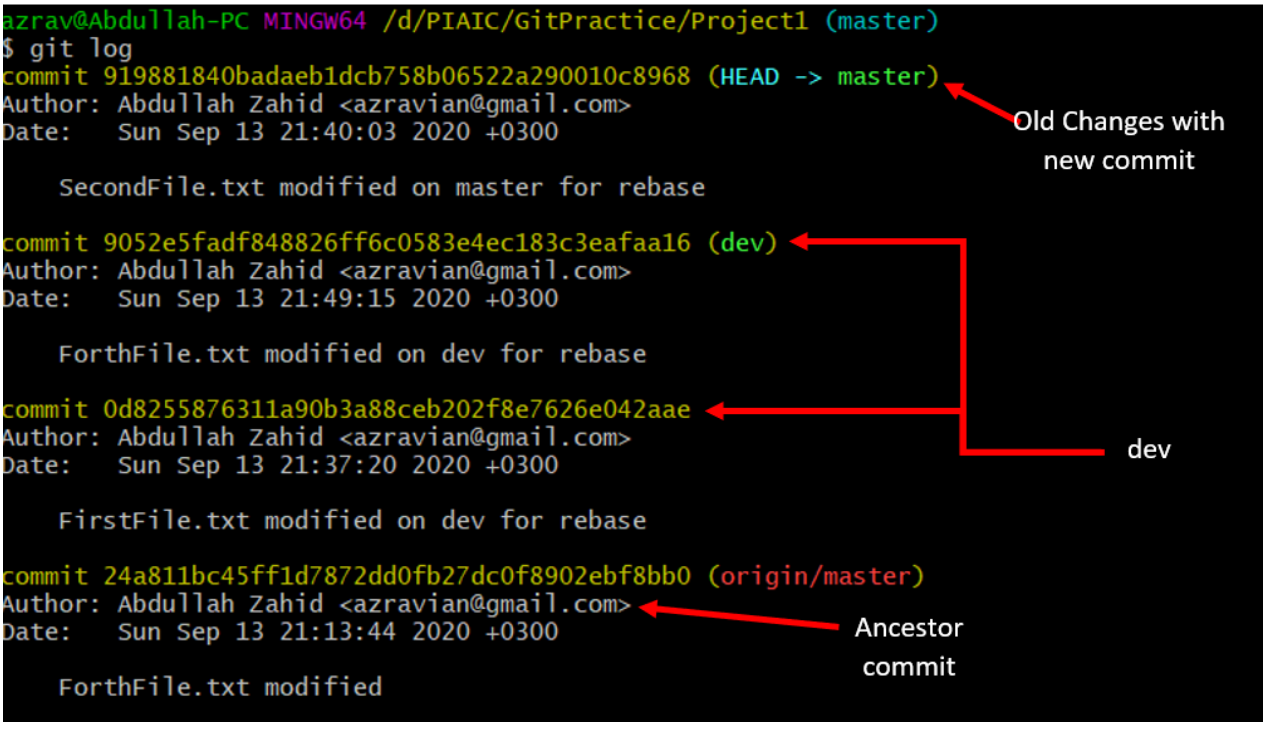
Logs from all branches:



A better representation can be seen by using graph:



Now let us rebase *dev* branch to *master*. After rebase the result is:



Notice that commit on *master* with hash value of *ed77e45* is no more in the history, and these changes are added with a new commit *9198818* with same message after commit from dev branch. The graphical view can give a better insight into the rebase.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
git relog --graph --all --date-order --decorate --oneline
* 9198818 (HEAD -> master) SecondFile.txt modified on master for rebase
* 9052e5f (dev) ForthFile.txt modified on dev for rebase
* 0d82558 FirstFile.txt modified on dev for rebase
* 24a811b (origin/master) ForthFile.txt modified
```

Submodules:

Submodules are nested repositories, which means these are repositories inside a repository.
There might be a case when a project, say Project-A have dependencies based on other project Project-B. While Porject-B is still evolving, We might need to keep this project up to date in Project-A. In such case, we can have Project-B as a submodule of Project-A. All operations such as push, pull, commit can be done to a submodule.

While working on submodules, we need to keep in mind:

1. Actual content of a submodule are not stored in parent repository.
2. Only local path, remote URL and checked out versions are stored in Parent repository.

Adding a submodule

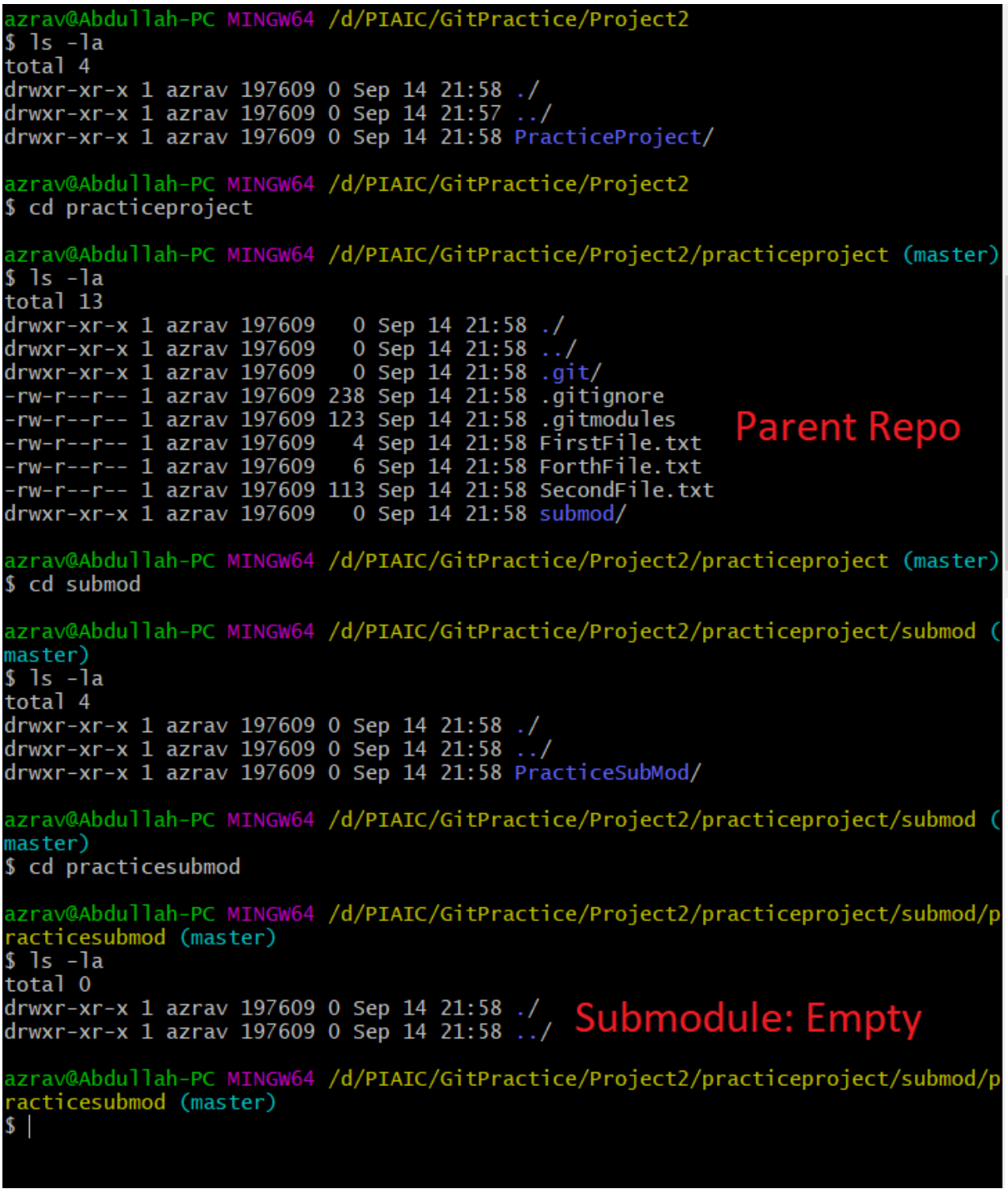
`git submodule add <SubModule_URL>`

This command initializes an empty sub module. for Example, I am adding an online reposiory and used following command (Please notice the directory paths in this section):

```
$ git submodule add https://github.com/azravian/PracticeSubMod.git
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod (master)
$ git submodule add https://github.com/azravian/PracticeSubMod.git
Cloning into 'D:/PIAIC/GitPractice/Project1/submod/PracticeSubMod'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory
```

As a submodule is added we can see that a file have been added in parent repository, with a name of *.gitmodule* and this file conains the information such as submodule local path and remote URL.



Besides this, git saves the record of submodule in local ".git/config" directory and also a copy of each submodule's .git is kept in its internal ".git/modules" folder.

After the addition of submodule, we need to commit this addition.

Cloning a project with submodule

Since the parent repository does not store the contents of submodule, cloning a parent repository does not clone the submodule. for our example, we had a submodule `PracticeSubMod` in parent repository `PracticeProject` . If we clone `GitPractice` from it's url using following command:

```
$ git clone https://github.com/azravian/PracticeProject.git
```

I won't get the content of `PracticeSubMod` . Let us create another folder and clone this repository using above command.

```
D:\PIAIC\GitPractice\Project2\PracticeProject>tree /f
Folder PATH listing for volume New Volume
Volume serial number is E827-AAE8
D:.
    .gitignore
    .gitmodules
    FirstFile.txt
    ForthFile.txt
    SecondFile.txt

└───submod
    └───PracticeSubMod
```

Cloning Submodules along with project:

Now if we wish submodule to be cloned with the parent repository we need the following command:

```
$ git clone https://github.com/azravian/PracticeProject.git --recurse-submodules
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice
$ git clone https://github.com/azravian/PracticeProject.git --recurse-submodules
Cloning into 'PracticeProject'...
remote: Enumerating objects: 229, done.
remote: Counting objects: 100% (229/229), done.
remote: Compressing objects: 100% (121/121), done.
remote: Total 229 (delta 72), reused 220 (delta 66), pack-reused 0
Receiving objects: 100% (229/229), 25.30 KiB | 602.00 KiB/s, done.
Resolving deltas: 100% (72/72), done.
Submodule 'submod/PracticeSubMod' (https://github.com/azravian/PracticeSubMod.git) registered for
 path 'submod/PracticeSubMod'
Cloning into 'D:/PIAIC/GitPractice/PracticeProject/submod/PracticeSubMod'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Submodule path 'submod/PracticeSubMod': checked out 'a0d4eb788fdf2e0d2c07f414c98ab10b4a3e1452'
```

Parent repo cloning

Submodule cloning

`recurse-submodule` makes git to clone all submodule in the repostiory. Now if we check the contents.

```
D:.
    .gitignore
    .gitmodules
    FirstFile.txt
    ForthFile.txt
    SecondFile.txt

└───submod
    └───PracticeSubMod
            First_in_SubMod.txt
            Second_in_SubMod.txt
```

We have got the content of submodules also.

Cloning Submodules after cloning project:

Suppuse we have already simply cloned a projetc using `git clone <ProjectURL>` command, submodules won't be cloned. Now if we wish to clone the we need to use following command:

```
$ git submodule update --init --recursive
```

Now I have cloned my project which have submodule, after clonning i didn't get the contents of submodule:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project2
$ git clone https://github.com/azravian/PracticeProject.git
Cloning into 'PracticeProject'...
remote: Enumerating objects: 229, done.
remote: Counting objects: 100% (229/229), done.
remote: Compressing objects: 100% (121/121), done.
remote: Total 229 (delta 72), reused 220 (delta 66), pack-reused 0
Receiving objects: 100% (229/229), 25.30 KiB | 996.00 KiB/s, done.
Resolving deltas: 100% (72/72), done.

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project2
$ cmd
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\PIAIC\GitPractice\Project2>tree /f
tree /f
Folder PATH listing for volume New Volume
Volume serial number is E827-AAE8
D:.
└───PracticeProject
    │   .gitignore
    │   .gitmodules
    │   FirstFile.txt
    │   ForthFile.txt
    │   SecondFile.txt
    │
    └───submod
        └───PracticeSubMod

D:\PIAIC\GitPractice\Project2>exit
exit
```

Now if I want those content, I need to run the above command:

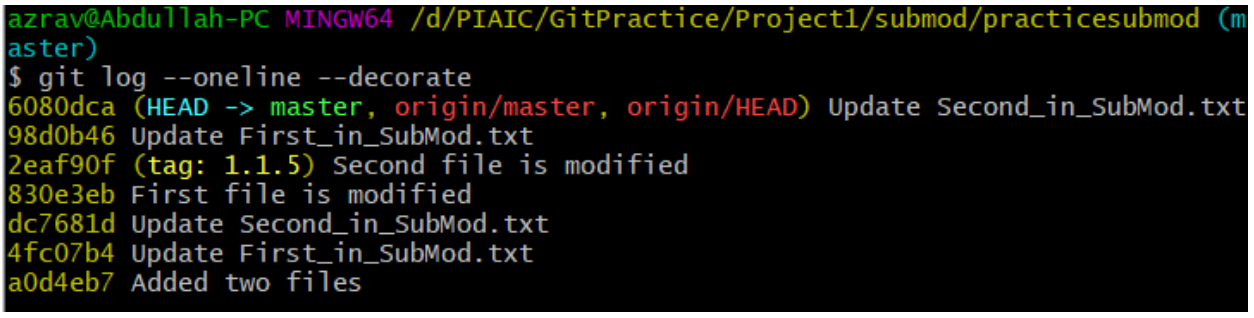
```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project2/PracticeProject (master)
$ git submodule update --init --recursive
Submodule 'submod/PracticeSubMod' (https://github.com/azravian/PracticeSubMod.gi
t) registered for path 'submod/PracticeSubMod'
Cloning into 'D:/PIAIC/GitPractice/Project2/PracticeProject/submod/PracticeSubMo
d'...
Submodule path 'submod/PracticeSubMod': checked out 'a0d4eb788fdf2e0d2c07f414c98
ab10b4a3e1452'

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project2/PracticeProject (master)
$ cmd
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\PIAIC\GitPractice\Project2\PracticeProject>tree /f
tree /f
Folder PATH listing for volume New Volume
Volume serial number is E827-AAE8
D:.
|   .gitignore
|   .gitmodules
|   FirstFile.txt
|   ForthFile.txt
|   SecondFile.txt
|
└───submod
    └───PracticeSubMod
            First_in_SubMod.txt
            Second_in_SubMod.txt
```

Checking Out Version of Submodule.

`git checkout <VersionTag>`

As it was discuss ealrier that the project can only use the checked out version of a submodule. Unlike the normal repository, the submodule always point to a specific version rather a branch. It is because branch can evolve over a time but a verion will not.

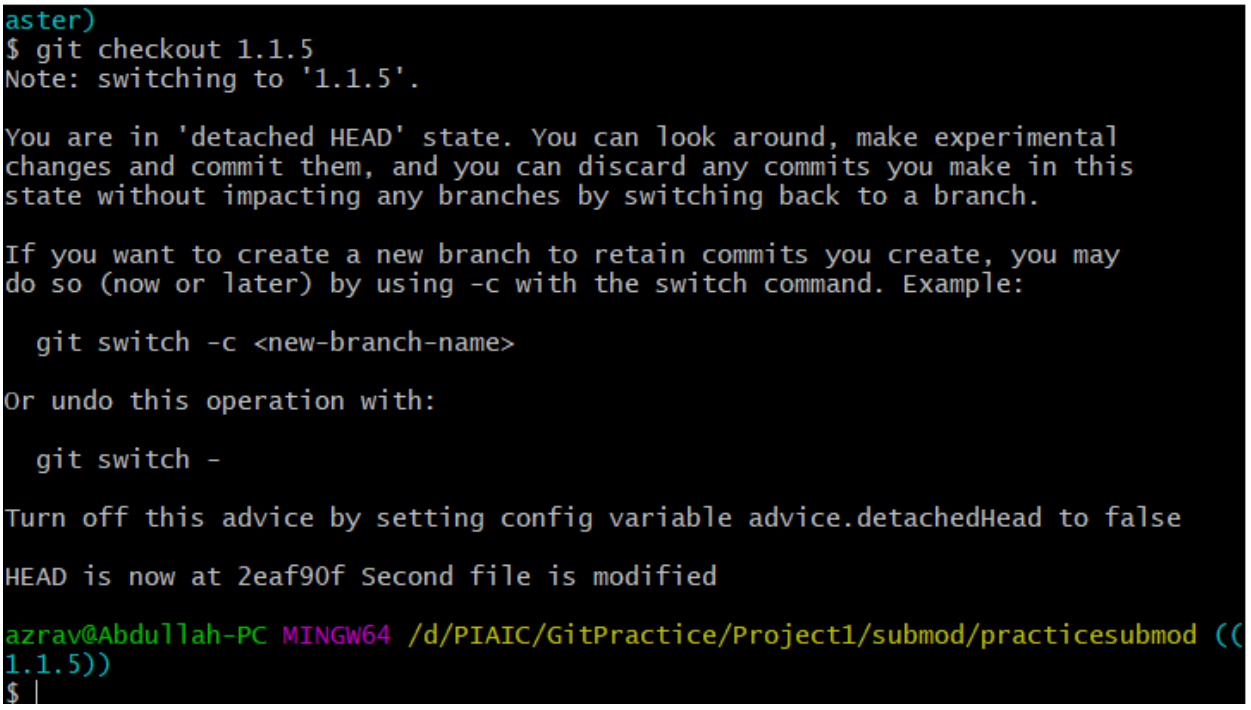
Now let us see logs of submodule, which are:



Notice a version with tag 1.1.5, we would like to checkout to this version:

```
$ git checkout 1.1.5
```

Now this version of submodule will be used in our project.



`git checkout <CommitHash>`

If we wish to checkout to a specific commit, we use the above command. suppose we want to checkout to the latest commit, that is 6080dca. we will use following command:

```
$ git checkout 6080dca
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod/practicesubmod ((1.1.5))
$ git checkout 6080dca
Previous HEAD position was 2eaf90f Second file is modified
HEAD is now at 6080dca Update Second_in_SubMod.txt

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod/practicesubmod ((6080dca...))
$
```

Notice that at the end of path, git bash shows the information about the checked out version of submodule. Once we check out to previous version running `git status` would show that we have new commits in submodules. It is due to the fact that we have moved to a previous verion. We need to commit these changes.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   submod/PracticeSubMod (new commits)

no changes added to commit (use "git add" and/or "git commit -a")

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git commit -am "Moved to tag 1.1.5"
[master 9973277] Moved to tag 1.1.5
 1 file changed, 1 insertion(+), 1 deletion(-)

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

In further notes, submodule checked out version will be `1.1.5` unless mentioned otherwise.

Checking the status of submodule.

`git submodule status`

This command shows the details about the current checked out version of git. Let us move to version with Tag 1.1.5 and then run this command in parent repository.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git submodule status
 2eaf90fb55c762fb0cc6d420b889f6e45149993c submod/PracticeSubMod (1.1.5)
```

Updating a submodule, when pointer is moved.

`git submodule update <path/to/submodule>`

Previuosly we had move our pointer to a previous version, Now suppose some other user have made changes and moved this pointer of submodule to other position. Now if we pull or merge those changes we would need to update the pointer in our submodule. Let us first pull those changes made by another user. The git shows that out submodule have been changed but in actual only pointer have been moved:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), 295 bytes | 9.00 KiB/s, done.
From https://github.com/azravian/PracticeProject
 * branch            master     -> FETCH_HEAD
   a9b48e1..b709787  master     -> origin/master
Updating a9b48e1..b709787
Fast-forward
 submod/PracticeSubMod | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

and if we check submodule's status

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git submodule status
+2eaf90fb55c762fb0cc6d420b889f6e45149993c submod/PracticeSubMod (1.1.5)
```

Notice the `+` sign before commit hash, it shows that the version recored in parent repository. Now it means that currently pointer is at version tagged 1.1.5 but the version in parent repository is different as it have been from remote and was moved by other user.

In this case we need to update out submodule, using following command. `$ git submodule update submod/PracticeSubMod`

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git submodule update submod/PracticeSubMod
Submodule path 'submod/PracticeSubMod': checked out '6080dcac978934455f2f45292bdc58e80bc32cdb'

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git submodule status
 6080dcac978934455f2f45292bdc58e80bc32cdb submod/PracticeSubMod (1.1.5-2-g6080dca)
```

Now our pointer have moved to the new position.

Checking for changes in Submodule.

In case we need to check if anything's been changed in submodule, we need to use fetch command. But first we need to move to submodule directory and then fetch the changes.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod/PracticeSubMod ((6080dca...))
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 710 bytes | 18.00 KiB/s, done.
From https://github.com/azravian/PracticeSubMod
   6080dca..a5975e1  master     -> origin/master
```

Getting changes in submodule from remote.

Once we know something have been changed and we want to get those changes, we simply pull those change using pull command:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod/PracticeSubMod ((6080dca...))
$ git pull origin master
From https://github.com/azravian/PracticeSubMod
 * branch            master     -> FETCH_HEAD
Updating 6080dca..a5975e1
Fast-forward
 First_in_SubMod.txt | 1 +
 1 file changed, 1 insertion(+)
```

while pulling in submodule we must specify remote and branch. At this stage if we use `git status` we get following status.

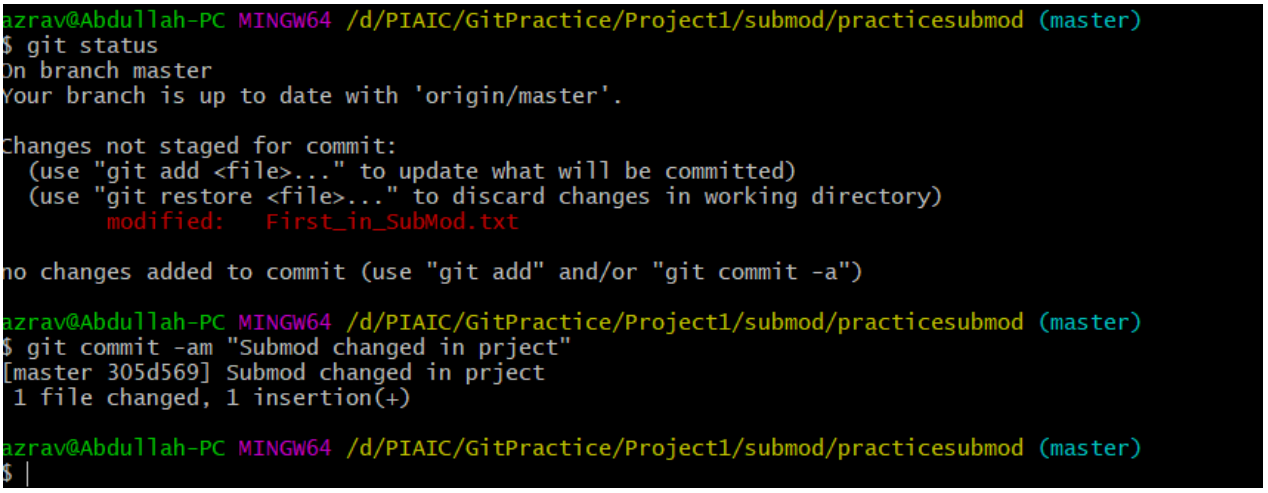
```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1/submod/PracticeSubMod ((a5975e1...))
$ git status
HEAD detached from 6080dca
nothing to commit, working tree clean
```

Even now we have a *Detached HEAD* as we moved pointer to previous version. If we wish to use the newer version that have just been pulled, we need to run following command:

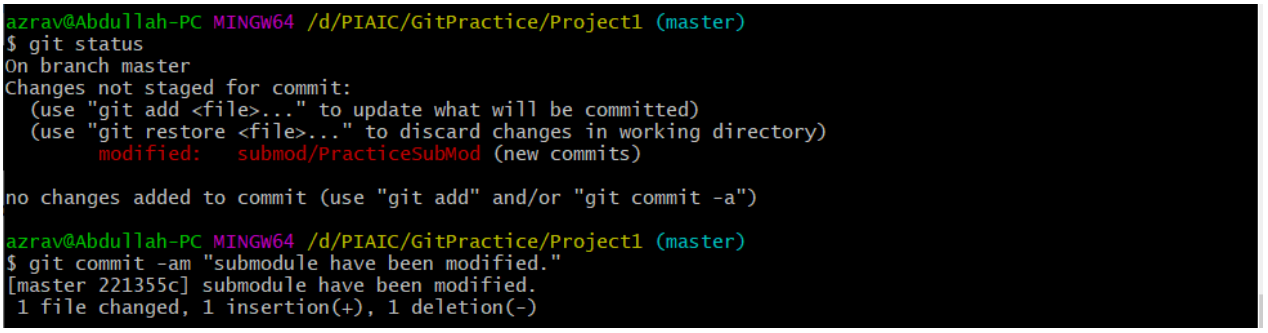
```
$ git checkout master
```

Making changes in Submodule.

In case we need to change something in out submodule, we need to commit those changes too. Since it is just a repository, the process is same as that of a normal repository. However one thing is to be kept in mind while making changes in submodules that we have checke out to some branch and not in detached HEAD. In case we are in detached HEAD, the commit will be gone as soon as we checkout to other branch or version.



Even though we have commit these changes we also need to commit these changes in parent repository.



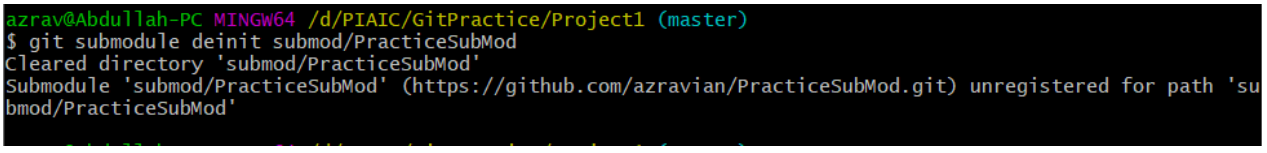
deleting a submodule

In case we need to remove a submodule from our parent repository we use following command:

1: `git submodule deinit <path/to/submodule>`

we need to run this command to remove all enteries from *.gitmodules* and config file for the specified submdodule. for our example:

```
$ git submodule deinit submod/PracticeSubMod
```



2: git rm <path/to/submodule>

Now run this command to remove the folder of submodule from git configuration. for our example:

```
$ git rm submod/PracticeSubMod
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git rm submod/PracticeSubMod
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory
rm 'submod/PracticeSubMod'
```

3: Delete directory:

Now just delete the directory of submodule.

Once all above steps are done commit your deletion of submodule.

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   .gitmodules
        deleted:    submod/PracticeSubMod


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git commit -am "submodule deleted."
[master cfd4062] submodule deleted.
 2 files changed, 4 deletions(-)
 delete mode 160000 submod/PracticeSubMod

azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$
```

Workflow with git-flow

When working in version control system, it is always crucial to agree on a workflow. If this workflow is not common, the confusion may arise.

If needed one can define own workflow or opt-out an existing one. One of such workflow is *git-flow*

git-flow:

git-flow provide us with handful of commands. Each of these commands performs multiple tasks automatically, which does simplify the version control.

git-flow isn't an alternative to git, but it just a set of scripts that combine standard git commands.

One of the most popular git-flow is *AVH Edition*. This git-flow is installed along with git.

Setting up git-flow:

`git flow init`

git-flow can be initialized by command given above. As git flow is initialized, a user can still use normal Git command or special git flow command. Once git-flow is intilaized, we need to use git-flow conventions. Let us start git-flow in our project by running the command above

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (master)
$ git flow init

Which branch should be used for bringing forth production releases?
   - master
Branch name for production releases: [master] master
Branch name for "next release" development: [develop] develop

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [D:/PIAIC/GitPractice/Project1/.git/hooks]
```

As soon as git-flow is initailized, we are prompted to ask for naming convention for our branches. Two major branches are *production release* and *next production release branch*, the details to be discussed later. The text enclosed in two *square bracket* is the default name or value for given branch. If we simplt press *enter* the default name will be assigned, otherwise a user may enter a name what he/she desires. I will go with the default names.

Branching model of git-flow

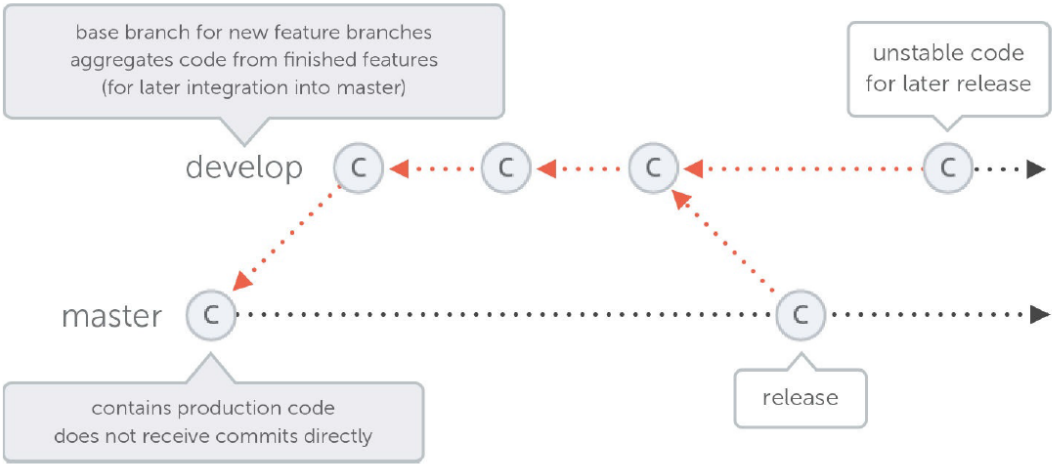
The git-flow has a versy specific branching model. And as mentioned earlier, There are two major branches in project, which tend to exist throughout the course of project, which are

`master` : The production branch

It always contains the code for production. No work is ever done directly on `master` branch. All work is done designated branch, such as feature branch or bugfix branch.

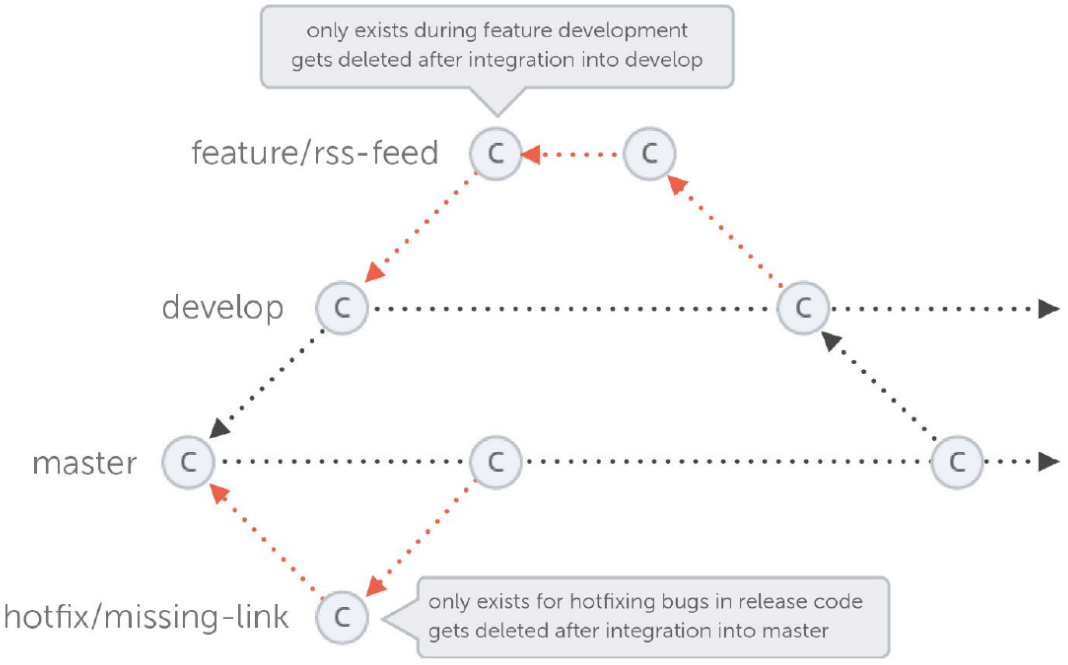
`develop` : Production development branch

This branch is dedicated for the effort for a new development. A new feature is always based on this development branch and as a feature is completed, it then merged to development branch. This branch combines all finished features and then deploy via *master*.



img ref: learn Verson Control with Git

Besides these two branches, there are many other branches, that have shorter life span. These branches have a very specific puspose. A depiction of these branches is:



`feature` branch:

This branch is dedication for each feature and exist only as long as work on that feature is being done. As soon as work is finished on this feature, the branch is merge to `develop` and then deleted. This branch name always have a prefix which we assigned during git-flow initialization, which for me is `feature/` .

starting a feature:

```
git flow feature start <FeatureName>
```

Suppose Someone wants to add a feature of FAQs on website, he would need to *start* the feature using above command and since we need FAQs

```
$ git flow feature start FAQs
```

and as soon as we run this command:

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (develop)
$ git flow feature start FAQs

Switched to a new branch 'feature/FAQs'

Summary of actions:
- A new branch 'feature/FAQs' was created, based on 'develop'
- You are now on branch 'feature/FAQs'

Now, start committing on your feature. When done, use:

     git flow feature finish FAQs
```

Notice the text enclosed in red box. The git-flow telling us about the steps it has taken after running the command. It has created this branch from develop branch and then checkedout to this branch so this one command is roughly a combination of following commands:

```
$ git checkout develop
$ git branch feature/FAQs
$ git checkout feature/FAQs
```

So roughly three commands are packed into one command.

Finishing a feature:

```
git flow feature finish <FeatureName>
```

Once we have done all work on this feature, we need to add this feature to our development and above command is used for this purpose. Let us assume we have worked on our *FAQs* feature and now we want to finish it.

```
$ git flow feature finish FAQs
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (feature/FAQs)
$ git flow feature finish FAQs
Switched to branch 'develop'
Updating feab499..e953f3b
Fast-forward
 FAQs.html | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 FAQs.html
Deleted branch feature/FAQs (was e953f3b).

Summary of actions:
- The feature branch 'feature/FAQs' was merged into 'develop'
- Feature branch 'feature/FAQs' has been locally deleted
- You are now on branch 'develop'


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (develop)
$
```

Now feature is merged in development branch. Now this feature can be tested and then released along with other features.

Now notice on the summary of the steps taken by git-flow, which are:

1. Feature branch merged into `develop` branch.
2. Feature branch is deleted locally.
3. checked out to `develop` branch.

Creating a bigfix

A bugfix if needed to solve a bug in development which isn't yet released.

starting a bugfix:

```
git flow bugfix start <BugFixName>
```

This command starts a bug fix in development. By default a bugfix is started from develop branch. let us start a bugfix:

```
$ git flow bugfix start fix125
```

```
azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (develop)
$ git flow bugfix start fix125
Switched to a new branch 'bugfix/fix125'

Summary of actions:
- A new branch 'bugfix/fix125' was created, based on 'develop'
- You are now on branch 'bugfix/fix125'

Now, start committing on your bugfix. When done, use:

     git flow bugfix finish fix125


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (bugfix/fix125)
$
```

The git have performed following steps:

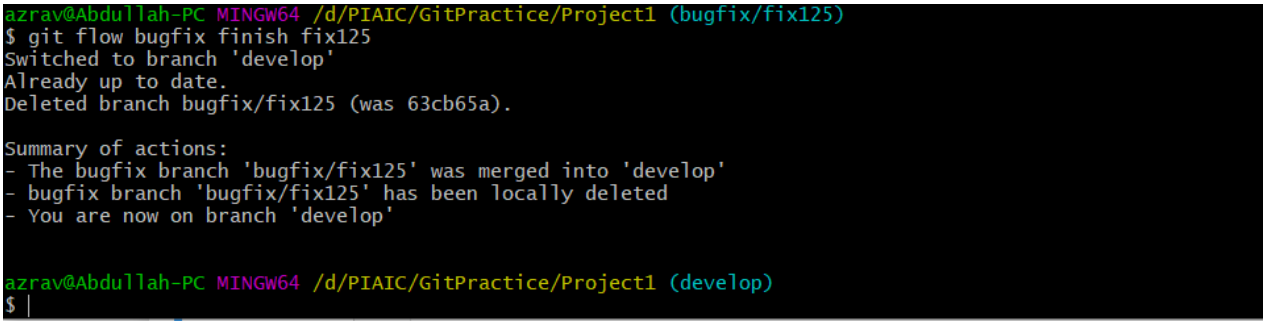
1. created `bugfix/fix125` from `develop` branch.
2. Checked out to `bugfix/fix125` branch.

Finishing a bigfix:

`git flow bugfix finish <BugFixName>`

This command finalizes our bugfix and icnorporate all changes back to our develop branch. for our example:

```
$ git flow bugfix finish fix125
```



after finishing bugfix, git does following steps:

1. Merge `bugfix/fix125` into `develop` branch.
2. Check out to `develop` .
3. delete `bugfix/fix125` branch.

Managing release

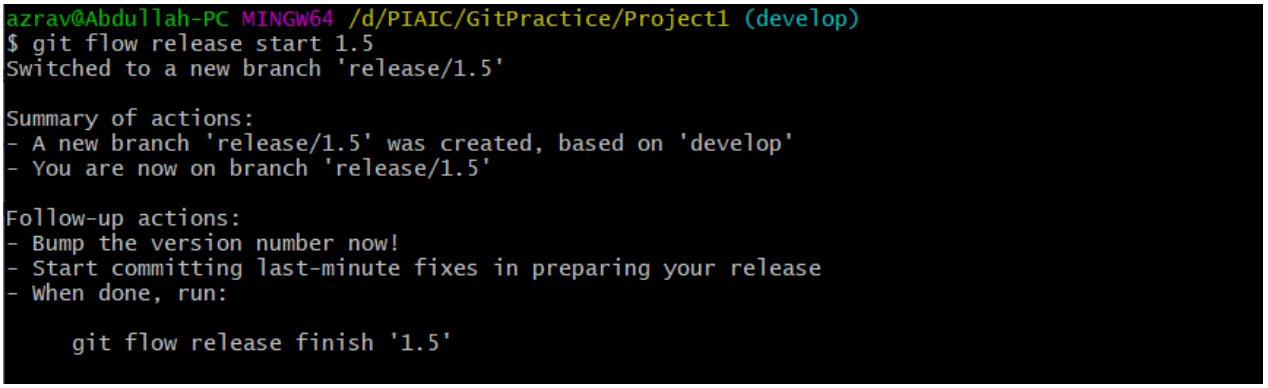
A release is basically addition of final changes into our production. Each release is identified by a verion number tag. A release is required after a bugfix or addition of feature.

Creating a New Release

`git flow release start <VersionNumber>`

Once we are done with all the features and extensive testing of these feature have been done, we might want to release these changes to production it is done by using above command. Now this command uses a verion number to specify a release and git will tag this release with this version number. Suppose we give a version number of 1.5:

```
$ git flow release start 1.5
```



At this stage, the git asks us for "follow up steps" which means if we still in need to make any change in our project.

Finishing a release:

`git flow release finish <VersionNumber>`

Once we are done with all changes we finish this release using above command using following command:

```
$ git flow release finish 1.5
```

```
Already on 'master'
Switched to branch 'develop'
Already up to date!
Merge made by the 'recursive' strategy.
Deleted branch release/1.5 (was e953f3b).

Summary of actions:
- Release branch 'release/1.5' has been merged into 'master'
- The release was tagged '1.5'
- Release tag '1.5' has been back-merged into 'develop'
- Release branch 'release/1.5' has been locally deleted
- You are now on branch 'develop'
```

Notice the steps taken by git when this command is run. which can be summarized:

1. Pull from remote to make sure everything is up-to-date.
2. Merge `develop` into `master` and `master` back into ``develop```.
3. Assign tag to this release. 1.5 in our case.
4. delete release branch and check out to `develop` branch.

Publish tags:

```
git push origin --tags
```

Now after pushing all changes to remote, we need to push our tags to remotes separately using above command.

Hotfix:

A situation may arise when we have some problems in the production and need to be taken care of. While we deal with such situation in development using *bugfix*, but for production we need *hotfix*.

Creating a hotfix:

```
git flow hotfix start <HotfixVersion>
```

this command will start a hotfix for any problem in production. Contrary to a bugfix, Hotfix is always based on `master` branch. A hotfix is always identified by a version number. Let us create a hotfix:

```
$ git flow hotfix start 1.3
```

```
$ git flow hotfix start 1.3
Switched to a new branch 'hotfix/1.3'

Summary of actions:
- A new branch 'hotfix/1.3' was created, based on 'master'
- You are now on branch 'hotfix/1.3'

Follow-up actions:
- Start committing your hot fixes
- Bump the version number now!
- When done, run:

     git flow hotfix finish '1.3'


azrav@Abdullah-PC MINGW64 /d/PIAIC/GitPractice/Project1 (hotfix/1.3)
```

This command does the following things:

1. create a branch `hotfix/1.3` based on `master`
2. check out to that branch

also notice that git asking for follow up steps which are:

Start committing your hot fixes

It means we now have to make changes and fix the problem.

Bump the version number now!:

This means that we need to update our verion number. It is because we are changing our profuction branch. And everytime we do this we should update our version number.

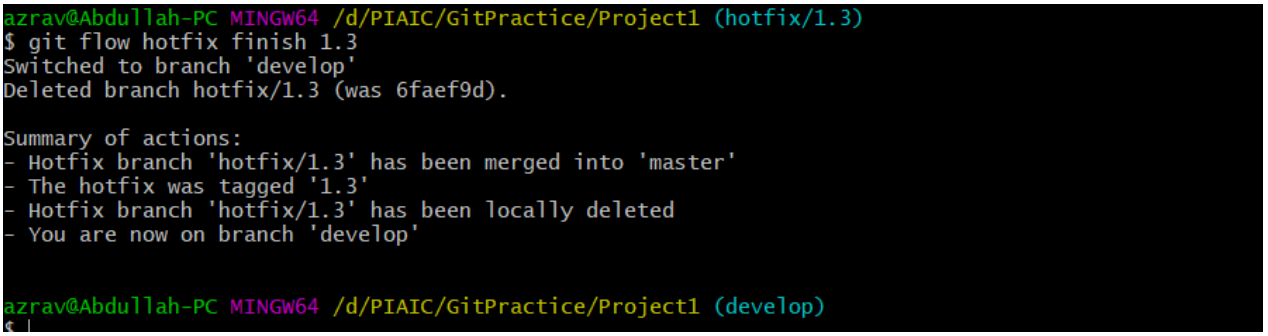
Finishing hotfix:

`git flow finish <HotFix Version>`

Now when we have done with all fixes, we need to run above command to finish a hotfix. for our example:

```
$ git flow hotfix finish 1.3
```

once this command is run the git asks for message and version number. After that hotfix is finished.



The git performed follwoing steps:

1. `hotfix/1.3` merged into `master` .
2. Tag assigned to hotfix.
3. Check out to `master`
4. Delete `hotfix/1.3` branch

after all this is done, it is good idea to push all changes and tags to remote.

git flow command which are not in book

`git flow feature start <FeatureName> [<Base>]`

This parameter enclosed in `[]` is optional. by default a feature branch is created from `develop` branch, but should we need to start a feature from some other branch we can specify that base branch. For example, we have a branch `Customer` and want to have a feature `Messaging` based ont this branch, We would use following command:

```
$ git flow feature start Messaging Customer
```

`git flow feature publish <FeatureName>`

if need to push our feature to remote during working on it we use command above.

`git flow feature pull <RemoteName> <FeatureName>`

This command pulls a feature from remote specified by `<RemoteName>`

`git flow feature track <FeatureName>`

Using this command we can track a feature from origin.

`git flow release start <ReleaseVersion> [<BaseCommit>]`

By default the release is done from the latest commit on development branch. However if we wish to do release from some previous commit, we can provide commit hash and start release from that commit. Suppose we want to start release from a commit with hash of *2f56e8b*:

```
$ git flow release start 2.5 2f56e8b
```

`git flow release publish <ReleaseVersion>`

This command pushes the relese to origin.