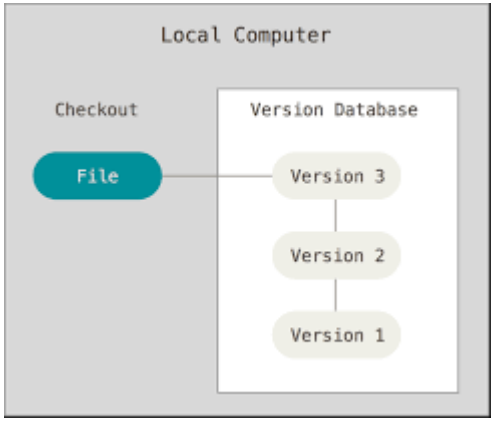
git and github

what is VCS ?

- V.C.S is version control software
- the V.C.S is used to track changes on any document
- It's also called Source code control software

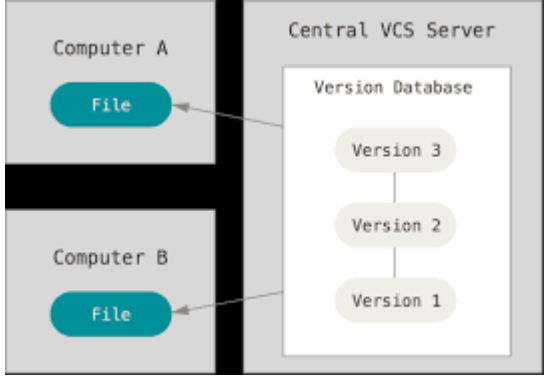
Types of version control

1. Local version control



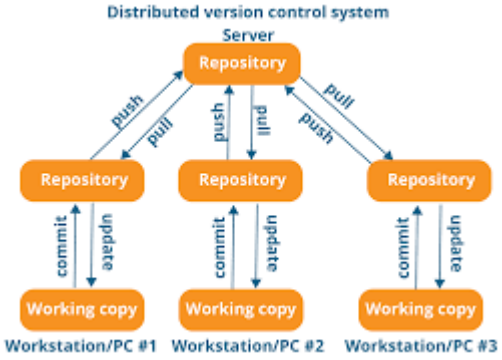
- local to the user on the PC

2. Central Version control



- working on a server for a business
- the source code is present on a central server
- the updates are live , No local copy

3. Distributed version control



- shared to group of people
- clone a copy of the code from the server

clone	download
bring the differences only	bring the whole project

- make downstream and upstream more easy

- make the project is in a live connection with the server when we need
- pull and push the modifications of the project
- you can connect to another project and have the access to all modification of both projects
- you can make your modifications locally and then push the modifications

what is so special about Git ?

- each version control takes the incremental difference in your code (incremental version control) --> due to disk space
- git takes a snapshot for all the file every modifications you make in the file

---

requirements for version control

- *the requirements is suited for the archetecture the developer build*

- track everything (content and metadata)
- OS independent
- unique ID , each object have unique ID
- track history
- no content change

the solution is to convert the files into objects in git

git tracking

- *git have objects*

- git objects is all things that git tracks
  - **blob** : file content + metadata
  - **tree** : folder content + metadata
  - **commit**
  - **tagged annotation**

OS independent

- git must be simple folder structure , also the file and the content
- can be read by any OS

- have a hidden folder called `.git` , contains all tracking information that can work on any OS (portable)
- `.git` is the same as `git repo` , that have the same commands for each OS

Unique ID

- we need an identifier uniquely to each project
- we make a module in git that generates hash function to encrypt the project and upload to github
- hash function take input x and process it ($f(x)$) then produce encrypt the input to produce ***unique*** output for each input
- using hashing algorithms --> SHA-1(secure hash algorithm 160bit , 256bit) , MD-5
  - testing the algorithm

```
echo "hello world" | git hash-object --stdin
```

- out `3b18e512dba79e4c8300dd08aeb37f8e728b8dad`
  - the same command in linux

```
echo "hello world" | shasum
```

- out `22596363b3de40b06f981fb85d82312e8c0ed511`

> **Note**
> **the 2 outputs are different why ?**
> - because the `git hash-object` take also metadata of the file like `type` , `size` and `null character`
> - then encrypt them

- lets see what git sees

```
echo -e "blob 12\0hello world" | shasum
```

- out `3b18e512dba79e4c8300dd08aeb37f8e728b8dad`

Track history

- git compares the last SHA-1 with the current SHA-1 to see if the content of the file has changed or not
- SHA : secure hashing algorithm
- when we change anything in the file , the SHA-1 differs from the last one

no content change

- in order to track a file we need to store tracks in an independent file not on the files we want to track

git architecture explain

- most of version controls have 2-tree architecture (`working tree` --> `repo`)
- but git have **3-tree architecture**

$$Working\ tree \rightarrow staging\ Area \rightarrow\ .git\ repo$$

- staging area == index == file

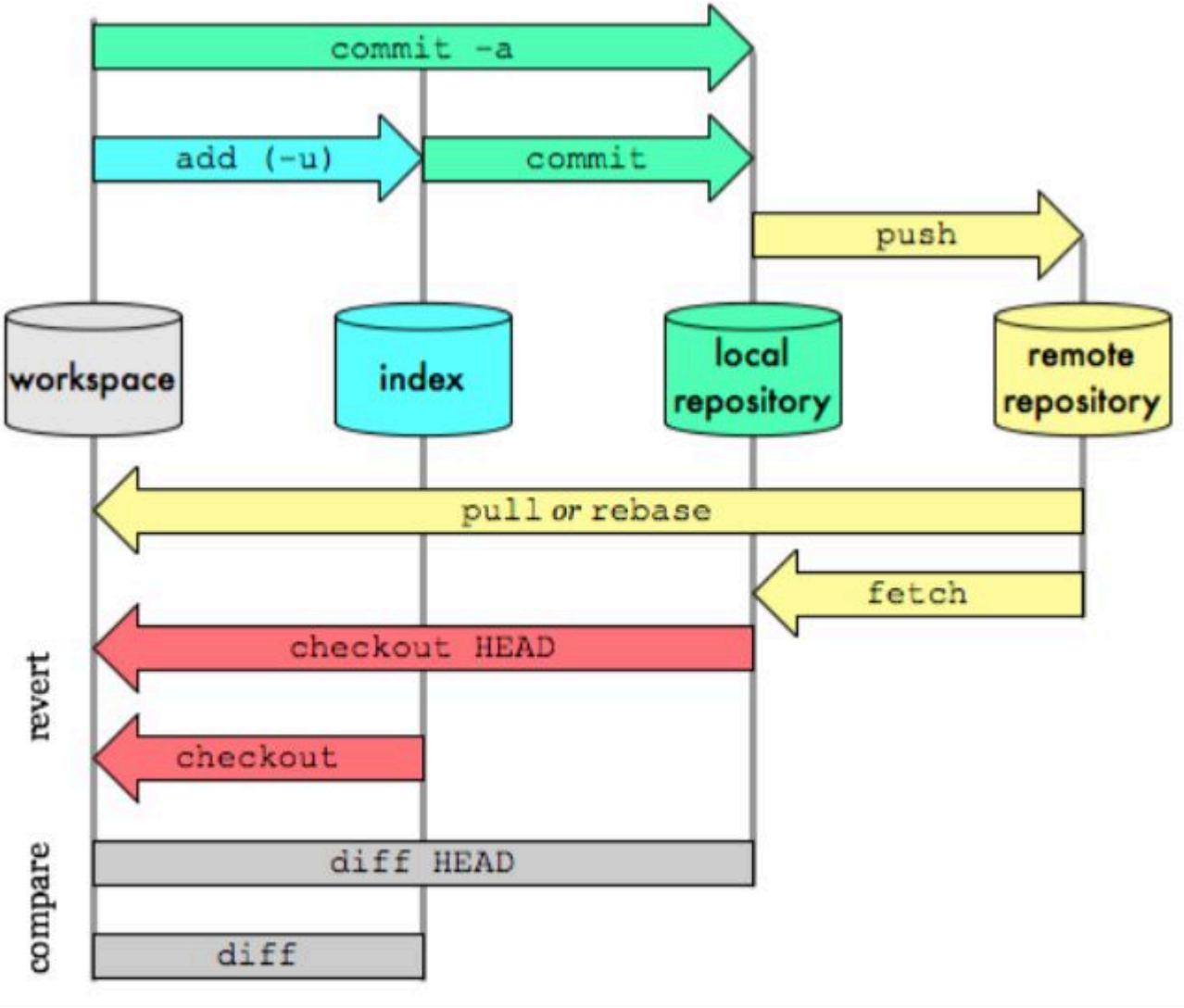
function of `Staging Area` ?

- we all know if we want to upload (commit) the changes , we will upload it directly on the repo
  - *problem* : we can have many , alot of versions and commits that will cause confusion when tracking

1. Staging Area function as a middle tier where U can stage alot of files before making the decision on which file will be committed , instead of making multi commits for multi files , make one commit for multi files
2. help in monitoring files before commit (monitoring workflow)
3. only commit the right files and making few mistakes

- we can make commit and staging at the same time

How does the 3-tier architecture work



- lets see , in the working tree we create a file
- this file is *untracked* file ---> it means that git doesn't know anything about the file , not stored in the `.git` file tracking
- git will give warning about the untracked file all the time
- the next move you should do is get this file tracked (move to the staging area)
- using the command `git add` we track the file

$$untracked \rightarrow tracked$$

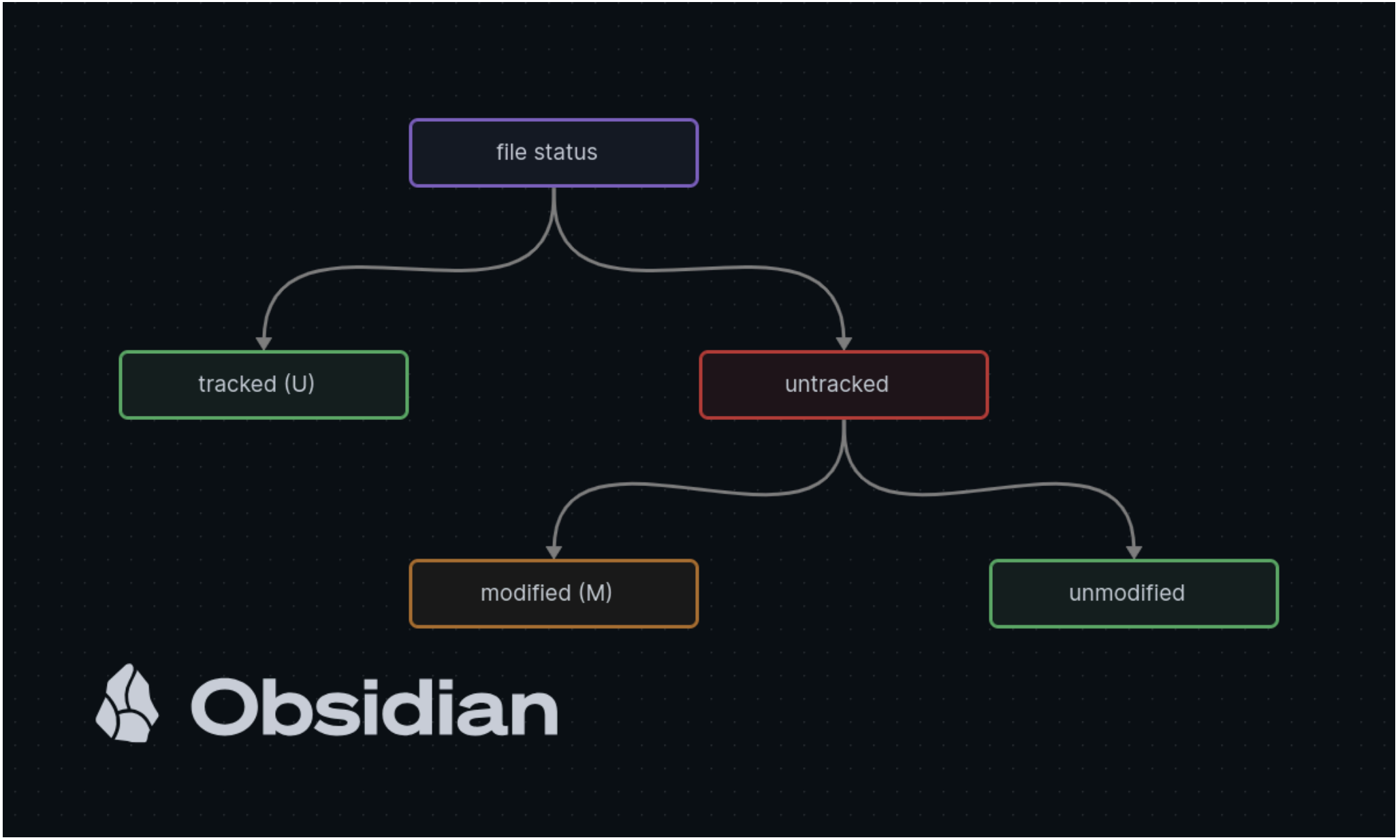
- **what happens when we add the file to the staging area (tracking the file)**
  - create a SHA-1 to the file in the staging area
  - also create a *blob* in the local repo have the name of the SHA-1 (used for tracking only)

- **Commiting changes**
  - after that , if the changes are approved then we commit it
  - **Commiting** : means that we add the changes from the staging area to the local repo
  - by using command `git commit` we commit changes to local repo

$$tracked \rightarrow commited$$

  - by now we have our first snapshot of the file changes

- if we made any changes after the commit the file status will change to *modified*



---

working with git

- before working with git , we need to configure the username and the email
- how can we do that

configuring the username and the email

```
git config --global user.name "Mostafa Samir"
git config --global user.email "Ur email"
```

see the configurations

- check if the configuration is set correctly or not
- display the user.name

```
git config --global user.name
```

- display the user.email

```
git config --global user.email
```

what is the meaning of `--global` ?

- it means that every project on this device will be set to the user.name and the user.email you assigned.
- if you want to assign the values of `user.name` and `user.email` to all devices (all of the system) we use `--system`

```
git config --system user.name "Mostafa Samir"
```

we can see all configurations

- we use `--list`

```
git config --list
```

- out `list of all configurations`

initializing git local repository

- we have to initialize git local repo in the working tree , in order to track all changes happened
- to do that we will use `git init` command

```
# initialize git local repo
git init
```

- this command will initialize `.git` file in the working tree

exploring git objects and trees

- after we initialize the `.git repo` file
- now the working tree is **local repo** in my device

- we need to see the status of the files in the local repo

files can be divided into

1. **Tracked** : added to the staging area
    - the tracked files can be divided into (t) A. unmodified: the version of the tracked files is identical of the version of the file now B. modified : the last tracked version is not the same as the current file (m)
2. **Untracked** : not added to the staging area (U)

- in order to see the status of the files we run the command `git status`

```
# see the status of the files
git status
```

See all files in the staging area

- we will use `ls-files`

```
# command to find all files in the staging area
git ls-files
```

```
# showing the SHA-1 for the file
git ls-files -s
```

See all files in the repo

```
# see files in the local repo
find ./git/object/ -type f
```

- git all objects present in `./git/object` that have type `f` or `file`

Adding file to index (Staging Area)

- we use command `git add`

```
# adding file1 to the index
git add file1
```

adding all files in the index

- we use the Option `*` --> wildcard

```
# adding all files
git add *
```

> **Hint**
>
> we can use regular expressions to control the adding of files

some notes

- when we add the files to the staging area , there are some changes

- we can see the that the index has stored the SHA-1 for the file , by using the command `git ls-files -s`
- we can see the `.git/objects` has stored the SHA-1 of the file as a blob , using the command `find .git/objects -type f`
- how to summarize the `git status` command to show summary --> `git status -s`

how the blob is stored in the objects file

- the first 2 letters of the SHA-1 is set for a folder name
- then the rest of the name is set for a file name (blob)

how to check that

- we can use the `cat-file` --> allow us to read the compressed files of git

```
git cat-file -t e90b994478d5d9a43602543de130cbb629a31a5f
```

- out `blob`

to see the size of the file

```
git cat-file -s e90b994478d5d9a43602543de130cbb629a31a5f
```

- out `86`

to see content

```
git cat-file -p e90b994478d5d9a43602543de130cbb629a31a5f
```

- out `# this is a python file for testing git`
  `print("mostafa is the greatest AI engineer")`

- all of this is base blob for indexing git use for tracking

> **pro tip**
>
> - Before commiting changes , when we run the command `find .git/objects -type f` we can see that git have the SHA-1 of the file that is staged
> - **this is not a snapshot** , it's only stored there for the tracking but it's not a version of the file and its not saved in the local repo

Committing changes

- we use command `git commit`

```
git commit -m "this is a message of the commit"
```

files in the local repo

- we notice that when we make a commit we have **3 SHA-1**
- and that's weird because we have only only 1 Blob

```
# see files in the local repo
find .git/object/ -type f
```

- out

```
.git/objects/4a/764b3c4e14c1fb157f4518b74643232b15c378
.git/objects/86/1d4590a9eb74198af112484f1899a8c02fbd43
.git/objects/e8/44cfc6921cdde965742f08fc57d2d2d6eb1460
.git/objects/e9/0b994478d5d9a43602543de130cbb629a31a5f
```

- the extra one is for file modification

lets track the changes

- every change in file will cause the change in 2 objects (**Tree** , **Blob**)
- so there is some fuzzy logic over there , how to know that both tree and blob are related to each other ?
- how to manage many changes in blobs and trees and find the relationship between them all

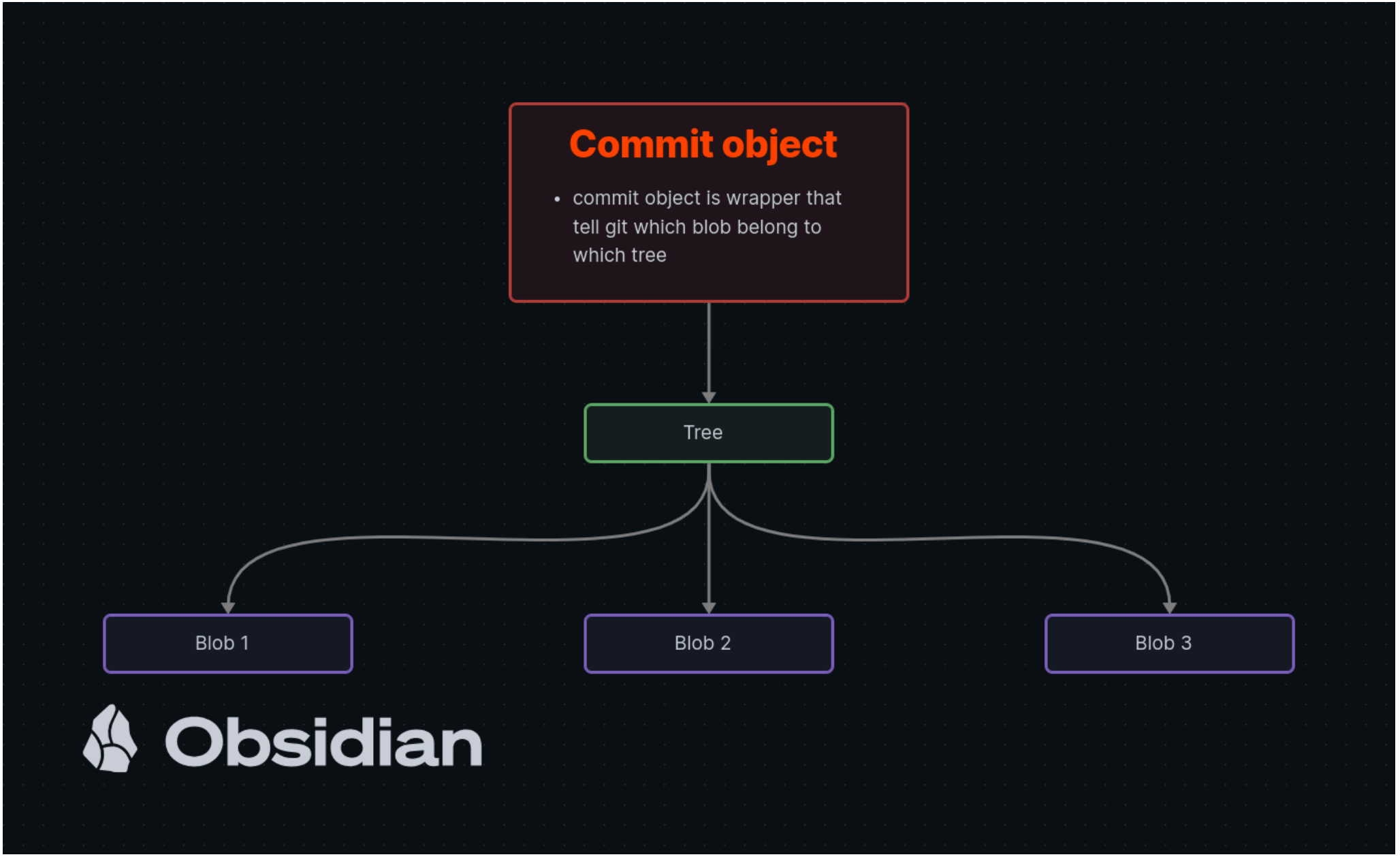
- so git have do add another object called `commit object`
- **Commit Object** work as a wrapper where it wraps both Blob and Tree that are modified
- **Commit Object** tells git that this Blob is modified inside this tree and they are together
- **Commit Object**
  - which Blob is modified / added
  - which Tree is modified / added
  - who modified / added it
  - when it was modified / added

- pointer to first modification / add

there are 3 objects in the repo

1. **Blob Object** : SHA-1 of the committed Blob
2. **Tree Object** : SHA-1 of the committed tree

3. **Commit Object** : Wrapper object



Basic git operations

committing changes

- we use `git commit` + option `-m` to pass a message

```
# passing a commit
git commit -m "this is a message"
```

Committing and adding in the same command

```
# add and commit at the same command
git commit -am "this is a message"
```

many commits

- when we add many commits the new commits have `parent` attribute

what is `parent` attribute ?

- parent attribute is added to the current commit that was changed to a new commit
- let's explain
  - you made a modification on a file
  - the file then have a different SHA-1 than the one in the `.git/objects` , so you need to add it and commit the changes
  - so the next logical step is to add the changes to the index and then commit them
  - after committing , **a new commit is added to the log** and also the new commit have **parent attribute**
  - ***parent attribute*** is a linked list that points to the old state of the commit the (old commit SHA-1) , this useful because it helps git to track all changes

- also have series of dates that is used in tracking the changes
- the first commit is called **root commit**
- the Linear series of commits is called ***Branch***
- the branch is by default is called **MASTER**

Working tree and commits

- git is able to track differences between working tree and commits
- the is a pointer called ***HEAD***
- if ***HEAD*** is pointed to the last commit , so the working tree is modified to the content in this commit
- the working tree is changed by changing the position of the ***HEAD*** pointer

differences

- finding the differences between the files
- we use `diff`
- difference between working tree and index

```
# finding the differences between files
git diff
```

- it will list all files that are modified and list all changes

> **NOTE**

`git diff` finds the difference between the working tree and the staging area
`git status -s` shows the difference between the staging area and the repo to get the same functionallity by the `git diff` command we use `--staged` option

```
# get differences between the repo and the index
git diff --staged
```

changing the commit message editor

- we can change the editor for the commit messages

```
# changing commit message editor
git config --global core.editor "nano"
```

making things easy with `git log`

having only one line

- git make it simple to have a report in one line

```
# having only one line
git log --oneline
```

HEAD

- **HEAD** is a pointer that point to the master branch version that is the same as the working tree
- if we moved this pointer to the previous version in the master branch , the content of the working tree will change

showing the modification of a file

- we use `git show`

```
# showing the changes in a file
git show _SHA-1_
```

- see the commit file , then see the tree then the blob , finally the content of the blob

differences between 2 commits

- showing the differences between 2 commits
- `git diff` --> the difference between the working tree and the staging area
- `git diff --staged` --> difference between the repo and staging area
- `git diff SHA-1..SHA-1` --> differences between blobs and trees in the 2 commits ***difference between 2 Commits***

```
# showing the differences between 2 commits
git diff 4a764b3..2778340
```

summarize the log

- we can summarize the log message in one line , by using `--oneline` option

```
# summarize the log message
git log --oneline
```

- also another way is to have all logs that is related to my file

```
# my file logs
git log --oneline file.txt
```

git show

- we can see the commit and its content , like mapping (not using `git cat-file -p`)
- we use `git show`

```
# mapping by using git show
git show c0847d9203a21b939feaa23d38b2940db5c10535
```

renaming file

- its preffered to use the git command for renaming file
- to avoid mistakes in git
- we use `git mv`

```
# renaming files
git mv old_name new_name
```

Deleting the repo

- if we want to delete all our work on the local repo on git we only have to remove the `.git` file

```
# deleting the local repo
rm -rf .git/
```

Undoing things

- if we made a mistake we need to have an Undo technique

un-staging file

- if we need to unstage a file from the index we will use `git rm --cached`
- used to stop tracking

```
# un stage a file
git rm --cached file1
```

Undo modification

- we can undo a modification in the working tree
- if you notice when you modified a file in the working tree , you can see that git noticed that the file in the working tree is not the same as the SHA-1 in the index
- we can undo the changes occured in the working tree and return to the latest file modification in the index

- git gives suuggestions about the command we can use to do that and that is `retore` command
- working tree <--- stagged

```
# undoing the changes happened in the working tree
git restore file1
```

- now the modification will be deleted from the working tree

restore modification from the local repo

- when we modify a file and add it to the index
- the file in the index is different from the file in the local repo
- what we need to do is

1. unstage the file

```
# un stage the file
git restore --staged file1
```

- this will cause the unstaging of the file

2. restore the changes

```
# restore previous changes
git restore file1
```

modifying the commit message

- we use the `amend` option

```
# changing the lst commit message
git commit --amend
```

- this will allow you to change the last commit message (modify)

returning to a previous commit

- we can do this by moving the **HEAD** pointer to the previous commit
- the head pointer points to the branch and the current working tree
- there are 2 ways to move the pointer
  - the first one is to move the changes to the staging area
  - the second one is to move the changes to working tree

explaining HEAD

- `HEAD` is a pointer that have a reference to a file
- this file contains the SHA-1 of the commit
- every time we make a commit , the SHA-1 is updated to the last commit SHA-1
- *lets see that*

```
# see the content of the head
cd .git
```

```
ls
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 cd .git
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo/.git$
 ls
branches        config       HEAD   index  logs      ORIG_HEAD
COMMIT_EDITMSG  description  hooks  info   objects   refs
```

- you can notice that there is a folder called `HEAD` , and that is the folder of HEAD pointer
- more discovering

```
# lets see what this pointer contains
cat HEAD ## we are in the HEAD file
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo/.git$
 cat HEAD
ref: refs/heads/master
```

- out ---> `ref: refs/heads/master`
- this is the value of the pointer , it refers to a folder called *ref --> heads --> master*
- inside the file **master** you will find the SHA-1 for the last commit

- going more deep , see the value of the SHA-1 in the master file

```
# go to file location
cd .git/ref/heads
```

```
# see file content
cat master
```

- out `222723655f2d450846601e77c87e5a8478aef944` <--- SHA-1 of last commit

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 cd .git/refs/heads/
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo/.git/refs/heads$
 ls
master
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo/.git/refs/heads$
 cat master
222723655f2d450846601e77c87e5a8478aef944
```

Moving `HEAD` forward and backward

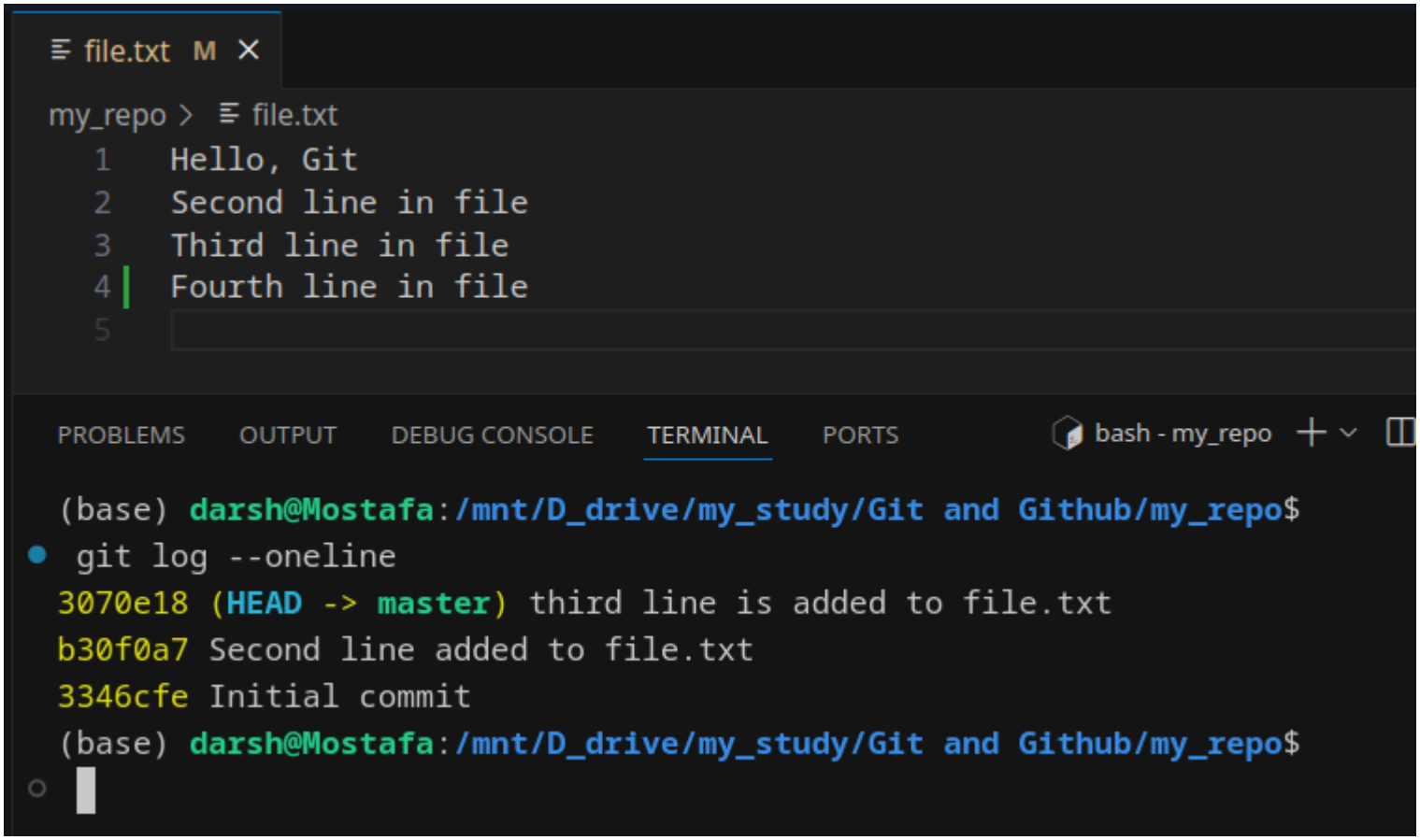
- we ca move the HEAD pointer forward and backward according to our need
- but be careful , any change in the head will cause a change in the local repo immediately
- the user choose if the changes occurred in the local repo can be stagged or applied to the working tree
- in both forward and backward we will use command `git reset`

Moving backward

- moving the HEAD backward is by using `git reset HEAD~1`
  - `~1` one move backward
  - `~n` n move backward
  - can also use order of the commit to move backward

```
# moving the head
git reset HEAD~1

# checking
git log --oneline
```

- out



Moving forward

- it's also called fast forward

- we use the order of the commits to make the HEAD at the last commit

```
# moving forward
git reset HEAD@{1}

# checking
git log --oneline
```

- out

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git reflog
3070e18 (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
2227236 HEAD@{1}: reset: moving to HEAD@{1}
3070e18 (HEAD -> master) HEAD@{2}: reset: moving to HEAD~1
2227236 HEAD@{3}: commit: Fourth line is added to file.txt
3070e18 (HEAD -> master) HEAD@{4}: commit (amend): third line is added to file.txt
2afe9a9 HEAD@{5}: commit: 3rd line is added to file.txt
b30f0a7 HEAD@{6}: commit: Second line added to file.txt
3346cfe HEAD@{7}: commit (initial): Initial commit
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git reset HEAD@{3}
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git reflog
2227236 (HEAD -> master) HEAD@{0}: reset: moving to HEAD@{3}
3070e18 HEAD@{1}: reset: moving to HEAD~1
2227236 (HEAD -> master) HEAD@{2}: reset: moving to HEAD@{1}
3070e18 HEAD@{3}: reset: moving to HEAD~1
2227236 (HEAD -> master) HEAD@{4}: commit: Fourth line is added to file.txt
3070e18 HEAD@{5}: commit (amend): third line is added to file.txt
2afe9a9 HEAD@{6}: commit: 3rd line is added to file.txt
b30f0a7 HEAD@{7}: commit: Second line added to file.txt
3346cfe HEAD@{8}: commit (initial): Initial commit
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git log --oneline
2227236 (HEAD -> master) Fourth line is added to file.txt
3070e18 third line is added to file.txt
b30f0a7 Second line added to file.txt
3346cfe Initial commit
```

tracking `HEAD`

- as we see if we run `git log` the last commit will be ignored and that is not convenient while tracking
- So , we use `git reflog` to track the logs of the HEAD

```
# track HEAD pointer
git reflog
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git reflog
2227236 (HEAD -> master) HEAD@{0}: reset: moving to HEAD@{1}
3070e18 HEAD@{1}: reset: moving to HEAD~1
2227236 (HEAD -> master) HEAD@{2}: commit: Fourth line is added to file.txt
3070e18 HEAD@{3}: commit (amend): third line is added to file.txt
2afe9a9 HEAD@{4}: commit: 3rd line is added to file.txt
b30f0a7 HEAD@{5}: commit: Second line added to file.txt
3346cfe HEAD@{6}: commit (initial): Initial commit
```

> **!!!**
>
> revert --> look for it

Tags

- a tag is marking a certain commit
- this is used to annotate a version of the file
- we simply use `git tag -option _version_` to make a tag
- the tag will be assigned auto to the last value of the `HEAD`

```
# making a tag
git tag -a v2.0.0 -m "Version 2.0.0 of the file"
```

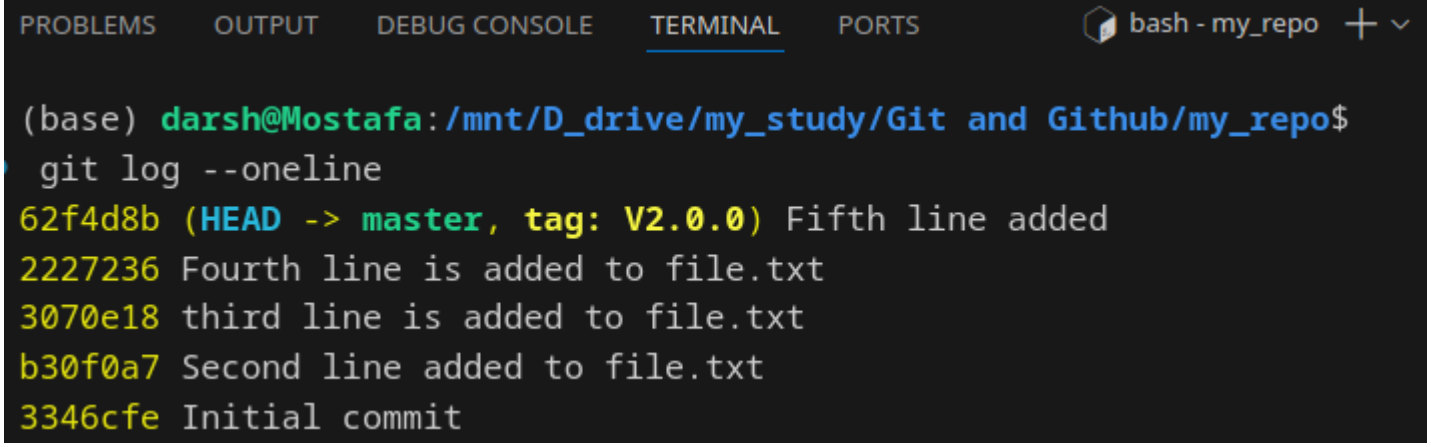
- now we can identify the versions from the normal commits
- how can we do that

```
# show the tag
git show V2.0.0
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git show V2.0.0
tag V2.0.0
Tagger: Mostafa Samir <mostafa.darsh.egy@gmail.com>
Date:   Thu Sep 26 17:42:51 2024 +0300

Second Version of the file

commit 62f4d8b1922e1917ddcfca98d634c3f628a510b2 (HEAD -> master, tag: V2.0.0)
Author: Mostafa Samir <mostafa.darsh.egy@gmail.com>
Date:   Thu Sep 26 17:41:39 2024 +0300

    Fifth line added

diff --git a/file.txt b/file.txt
index 52e1b49..f19d56b 100644
--- a/file.txt
+++ b/file.txt
@@ -2,3 +2,4 @@ Hello, Git
 Second line in file
 Third line in file
 Fourth line in file
+Fifth line --> version 2.0.0
\ No newline at end of file
```

- check the log files

```
git log --oneline
```



> # !!!
>
> tags differences

Git Branches

- git stores the blobs under trees and all of that is wrapped in a commit object , then the SHA-1 is created
- if we modify anything , the SHA-1 will change
- series of commits is called **Branch** (Linear Series)

- you can have many branches in the same project
- you may work on a feature but you to not want to make changes to the main branch , so we need to have a branch to work on instead of the main branch

- the branching provide non-linear development

Making a new branch

- we can make a new branch by using `git branch`

```
# making a new branch
git branch test1
```

> **Note**
>
> - the new branch is created where the `HEAD` is located

See which branch you are working on

```
# see which branch Iam working on
git branch
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git branch
* master
  test1
```

Switching through branches

- we can switch through branches by using `git switch` or `git checkout`

```
# switch to test1 branch
git switch test1
```

- now any change will be apply on the new branch **testing branch**

- after we finish changes , we merge branches if the changes is approved

Merge branches

merge

- we can merge 2 branches by using `git merge`
- note that if we want to merge on the master branch we will switch to the master branch and then use the `git merge` + *the other branch name*

```
# switching to the master branch
git switch master

# merge the other branch with the master
git merge test1
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git merge test1
Updating 45d8a23..9ec3e38
Fast-forward
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

Check the merging

- if we want to check the merging branches , we use `git branch --merged`

```
# checking the mergeed branches
git branch --merged
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$ git branch --merged
* master
  test1
```

- if there is/are branch/es that is not merged , the `git branch --merged` will act differently

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$ git branch --merged
* master
  test1
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$ git branch
* master
  test1
  test2
```

- in the above image , the branch test1 and master branch are merged together , while branch test2 is not merged with any other branch

deleting merged branch

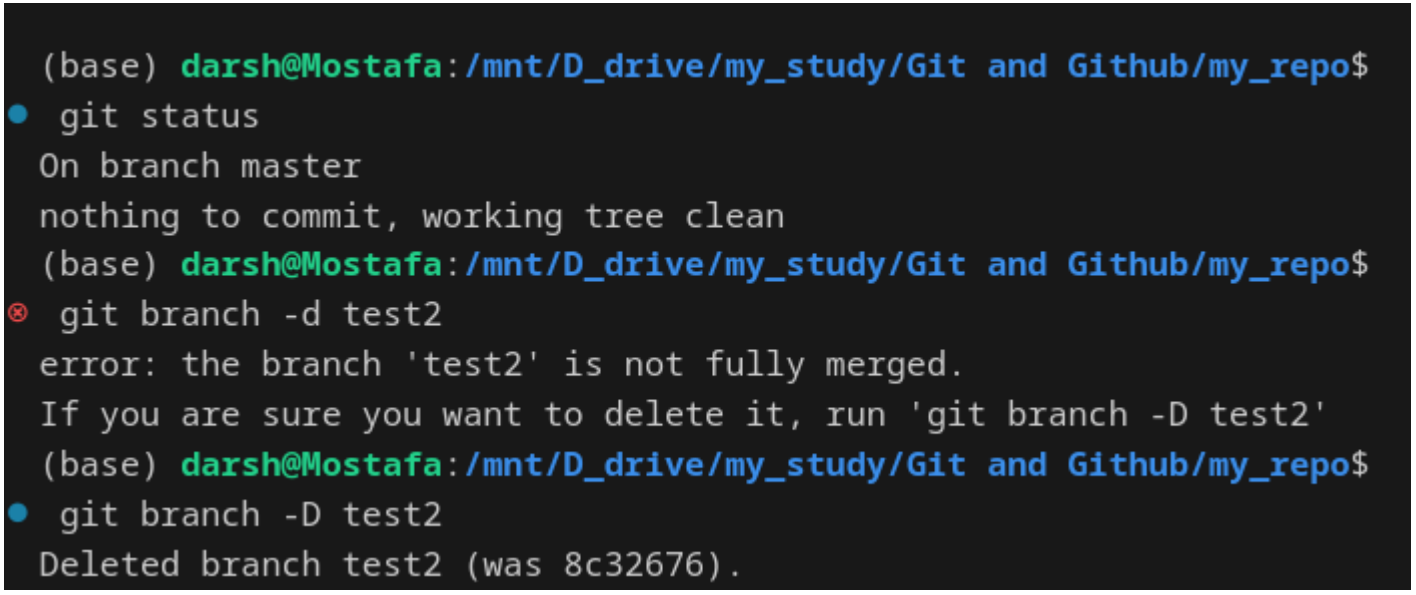
- since we have merged the test branch now we can delete it

```
# deleting a branch
git branch -d test1
```

deleting by force

- if the branch you are working on have changes that is not committed to the master and you want to delete it
- git will prevent you if you use the option `-d` instead use `-D`

```
# deleting branch by force
git branch -D test2
```



Branches variations

- we can apply some changes to a branch like testing something in the test branch and then if the changes is approved we can merge the 2 branches together

- how can we track the changes visually in both branches
- will , it will require a bit big command

```
# visually tracking the branches
git log --oneline --graph --decorate --all

# making an alias of the command
git config --global alias.graph "log --oneline --graph --decorate --all"

# using the alias
git graph
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/my_repo$
 git graph
*   da777f0 (HEAD -> master) Merge branch 'test1'
|\
| * b015d43 (test1) fifth line is added in branch test1
* | bed0cb1 new file is added in master branch
|/
* daaf12b first commit in testing branch
* bec5700 Third line added
* 12997a7 Second line added
* 472c9c7 First line is added
```

- **Explanation**
  - we made a new branch called *test1*
  - we added some changes to the file in the *test1* branch
  - then added a new file in the *master* branch
  - after the changes was approved , we merged the branches using `git merge test1`

- so that in the git file a new commit is created `da777f0` and this type of mergeing called **Three way merging**
- you noticed that its NOT fast forward merging , because we don't have a straight line to follow
- the **Three way merging** uses the last commit in the master and both new commits in master and test then make a new commit --> *merge commit*

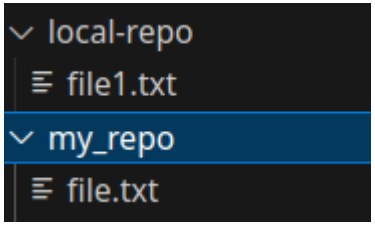
Working with Remotes

- the main function of git and github is you can work with remote repositories without having a live connection between the server and the local repo

Creating remote repo

- at the moment we will create a remote repo on my device

```
# create a repo
git init
```



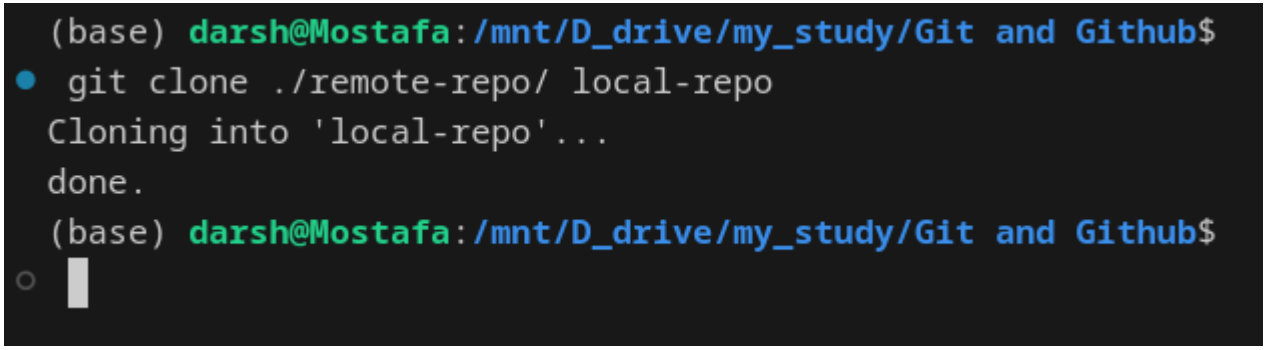
Cloning the remote repo

- cloning is to get all the remote repo on your device
- cloning is so different than downloading

cloning	downloading
getting all updates of the file and git information	getting all updates of the file onle
get all git history tracking , files , commits	none of git fies

- syntax ---> `git clone` + `URL` + `new name` (optional)

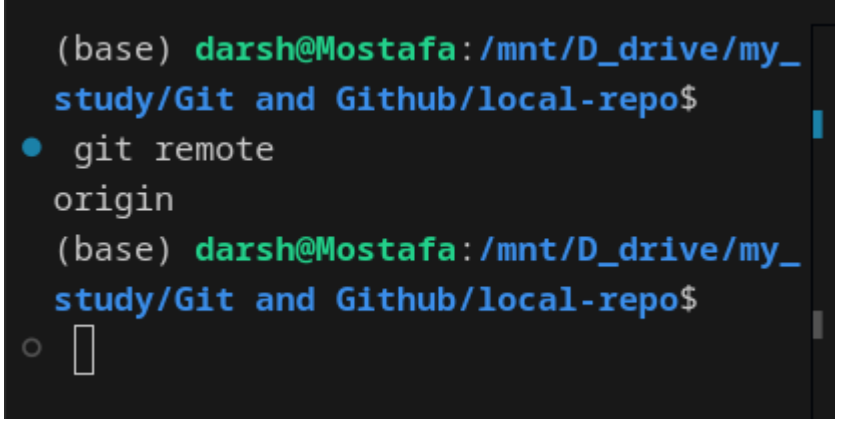
```
# cloning a repo
git clone ./remote/ local-repo
```



git remote

- it show info about the remote repo that was cloned

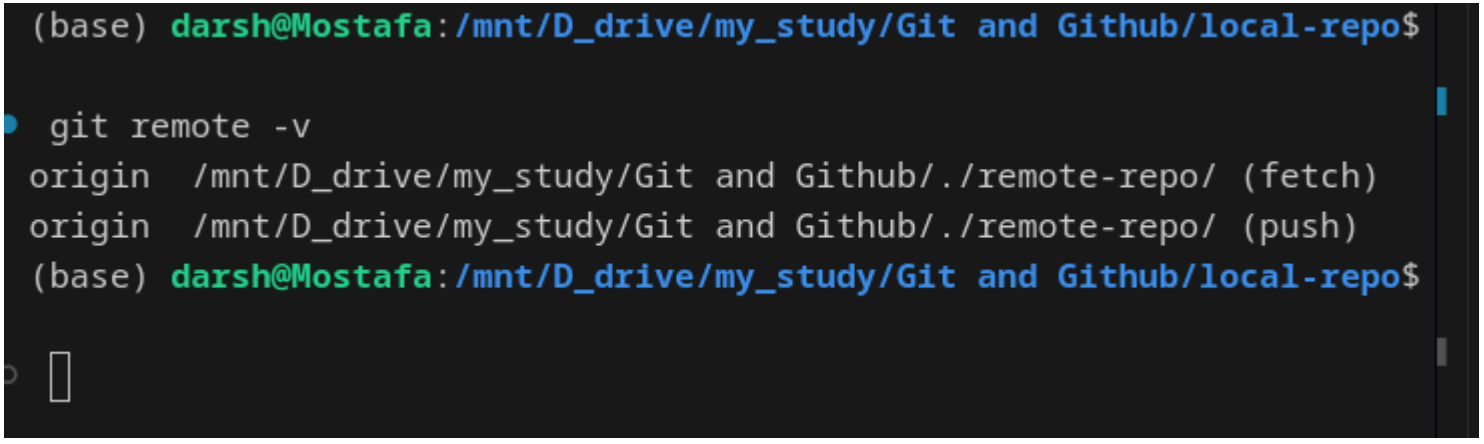
```
# info
git remote
```



- origin is the short name for the remote

- to show more info we use `-v` option

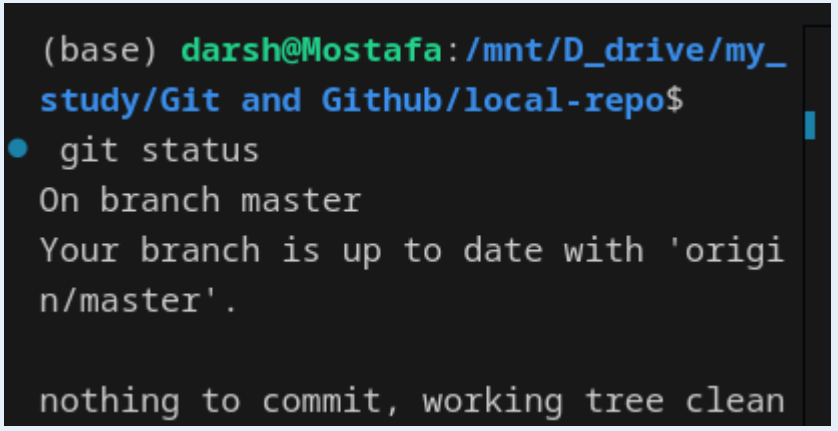
```
# more info
git remote -v
```



- it shows more options like `fetch` and `push`
  - `fetch` : cloning the updates from the remote repo
  - `push` : uploading the updates from the local repo

Note

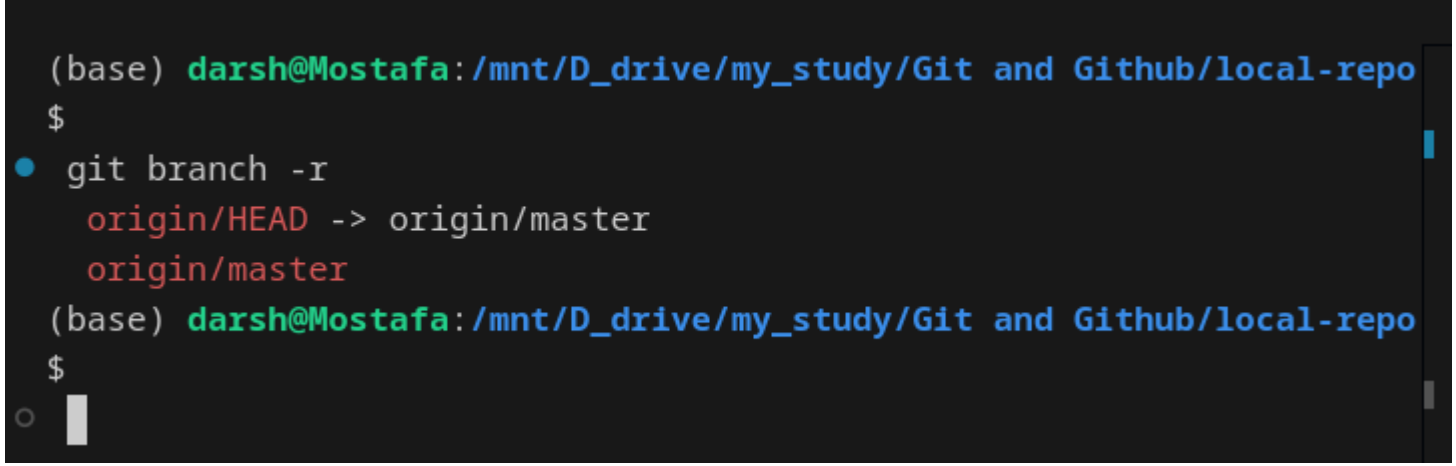
- if we changed the content of a file in the remote repo and commit it , the local repo will show you that you are up to date and that is not true
- because you didn't fetch the updates of the remote repo yet
- that why we use `git fetch`



showing branches in the local repo

- will `git branch` is ok , but we will use the option `-r`

```
# show the branches on the local repo
git branch -r
```



- it show that the origin is at the same point as the master branch in the local repo
- the origin HEAD points to this point

git fetch

- fetch is used to get the updates from the remote repo , BUT not merging it to your working tree
- it's used to alert you that there is updates that are in the remote repo and if U want to added it you can proceed

```
# fetching updates
git fetch
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 260 bytes | 260.00 KiB/s, done.
From /mnt/D_drive/my_study/Git and Github/./remote-repo
   b85c7ec..4c9f10e  master     -> origin/master
```

- now if we run `git status` things will change

```
# see the changes fetched
git status
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-fo
rwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$
```

- now we have 2 ways to catch this update
    1. use `git merge`
    2. use `git pull`
- in this case we used `git merge`

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git merge
Updating b85c7ec..4c9f10e
Fast-forward
 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

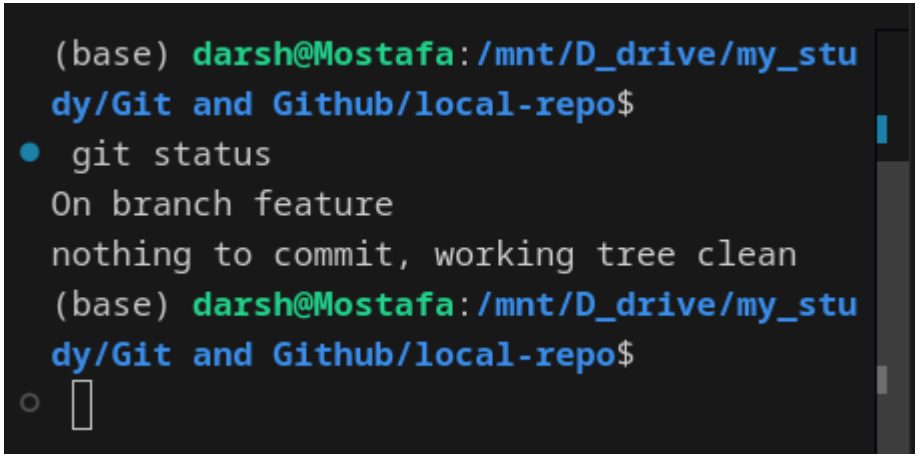
- when committing changes to our local repo , now we are ahead of the remote by a commit

```
(base) darsh@Mostafa:/mnt/D_drive/my_stu
dy/Git and Github/local-repo$
 git status
On branch master
Your branch is ahead of 'origin/master'
by 1 commit.
  (use "git push" to publish your local
commits)

nothing to commit, working tree clean
(base) darsh@Mostafa:/mnt/D_drive/my_stu
dy/Git and Github/local-repo$
```

- we can push changes by this point

Normal workflow

- in the normal workflow , we usually don't modify on the master branch
- we create a branch from the master and then we modify this branch then when we finish we push this branch to the remote repo
- lets proceed --->--->--->---> **LOCAL REPOSITORY**
  - creating a branch
  - commit on it
  - see the status



- well this is weird because we have a branch called feature , we made a commit but in the status we have no changes
- that's because we don;t have a branch on the remote repo , so git can compare between the two of them
- git don't understand what is the relationship between the 2 branches
- to prove this point we will try to push the changes

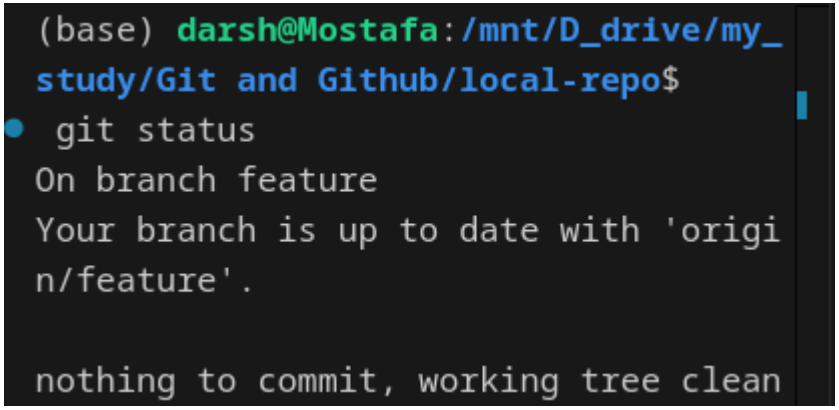
```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git push origin
fatal: The current branch feature has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.

(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$
```

- to solve the problem we will use the recommended command `git push --set--upstream origin feature`

```
# pushing the feature branch to remote repo
git push -u origin feature
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git push -u origin feature
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 300 bytes | 300.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To /mnt/D_drive/my_study/Git and Github/./remote-repo/
 * [new branch]      feature -> feature
branch 'feature' set up to track 'origin/feature'.
```

- let's see if the problem is fixed



pull changes

- `pull` is different than the `fetch` command
- pull command is `fetch` + `merge`

```
# pulling changes
git pull origin
```

```
(base) darsh@Mostafa:/mnt/D_drive/my_study/Git and Github/local-repo$

 git pull origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects:  50% (1/remote: Compressing objects: 100
% (2/remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 279 bytes | 279.00 KiB/s, done.
From /mnt/D_drive/my_study/Git and Github/./remote-repo
   4c9f10e..c83a72c  master     -> origin/master
Updating 4c9f10e..c83a72c
Fast-forward
 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

!!! pull from multi repos

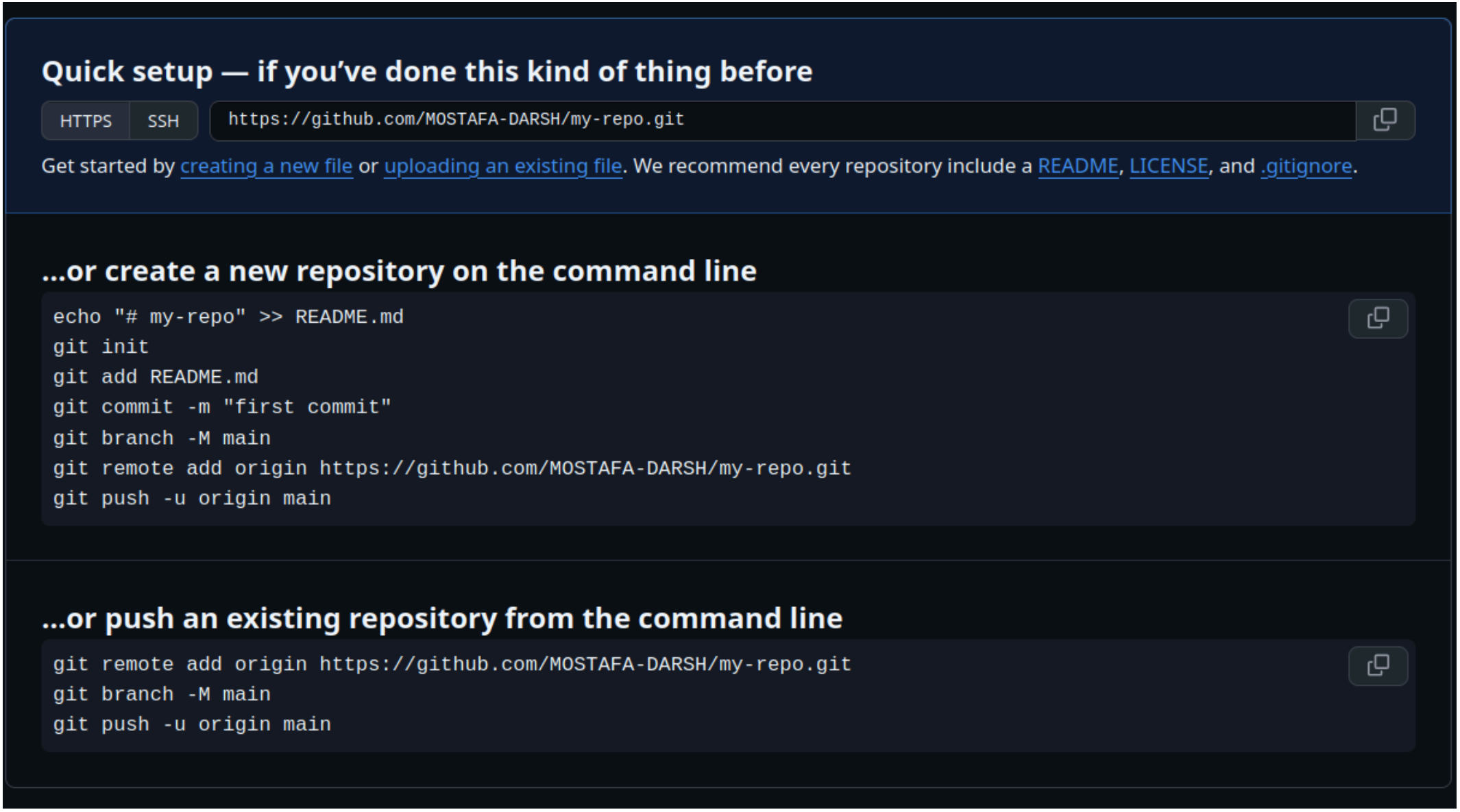
Github

- github is a hosting website that can host your repos and share it

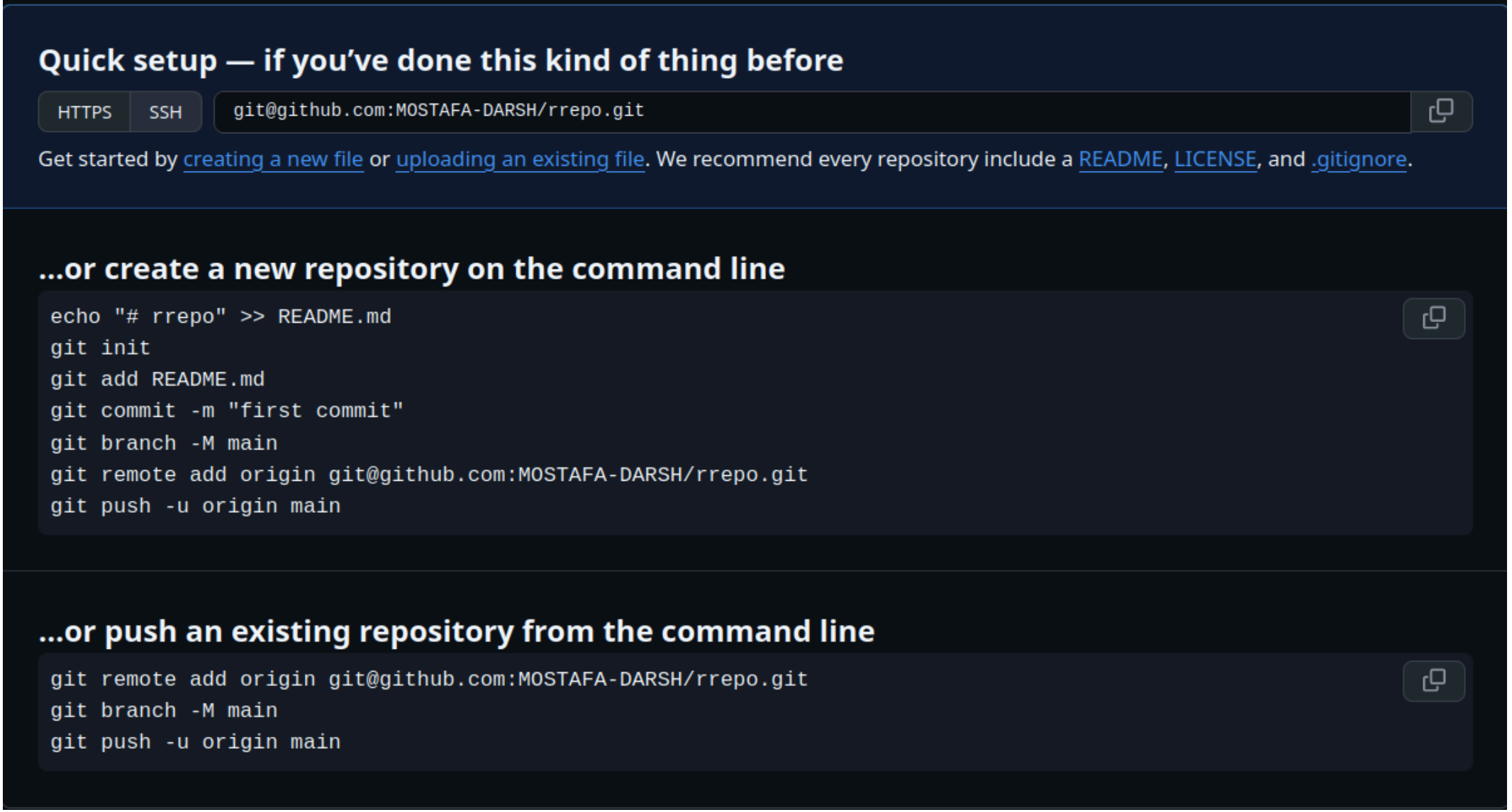
Workflow in github

- we create an empty repo
- added it to the local (attaching)
- push changes

attaching using https



attaching using SSH



Fork

- fork the repo ---> copy the git hub repo (foreign) to my github
- the same config will be copied
- you can work on this repo then contribute in the foreign repo

- **Workflow**
  - make a fork from the repo I want
  - clone the fork I copied (clone the fork not the original repo)
  - after adding changes , add remote repo of the original repo
  - add changes on the repo
  - push on the fork I copied
  - then If you want to contribute , make a pull request

pull request

- pull request is used to pull changes you have made on the fork to the original repo you forked
- you make a pull request from github and write a message for the owner of the original repo
- the owner have the choice to accept or deny

alias function script / bash scripting convert markdown to html file

points to look for

- merging conflict
- rebase
- types of tags