Git

Version Control System == Source control manager

FF X rollback

VCS is also essential for one of the 3 careers, even for a network engineer(as one of them made the configs of HW in files, so that to be able rollback in case latest version is down.)

LVC	CVC	DVC
	you don't have local copy. you're just working live on the central server(e.g. data center).	you clone(NOT download) the project from the server, then after modification you can get differences occurred since you cloned it, then push it (I): as you didn't lose connection, but the modification is just local. After the author watch your version, he admits it by applying merge. Also, you can deal with another user's version in the SAME time.
		Git is here !

SW/data architect thinks of the features as solutions to preexistent problems, not just for luxury.. That is the methodology in this tutorial. شخبط وانت مطمن

Incremental(Differential) VCS, like compiling C source codes.. Git isn't, but takes the snapshot of the file entirely.. previously the storage wasn't that big nowadays so it was restriction to do as Git does.

Git Architecture:

Each Architecture has requirements, not just OK!

> Requirements of Git VCS:
>
> 1. Track everything: content+metadata
> 2. OS independent, portable.
> 3. Unique ID for any object being tracked (file/folder)
> 4. Track History: not just the date, but the creation order
> 5. No content change: properties of the object are to be RO(Read-Only)
>   Working directory == Working tree

the format saved -in the repo- of the file/folder being tracked mustn't be the same as the original one.

1. Git objects:
   blob --> content
   |> metadata(size, permissions, type, and name)
   tree --> content
   |> metadata
   commit
   tagged annotations
2. The structure is merely is simply a folder(not to be complex as a database), and also encryption is not necessary, while compression is good for read speed
   .git folder: is the repo of the project, as a hidden folder, transforms your project dir into *repo*

> by this folder, everyone with the link have access to all versions.

3. files/dirs are no longer with its original ID 4. It is to have a module in our repo, generating hash for each object

NOTE: `git hash-object` reversly appends the content with: type size\0, before applying the hash function, so you experience difference in the output w.r.t `hash` command..
but keep in mind that only a SPACE affects the hash output.

Type of Git arch.: 3-tree architecture

compared to previous VCSs, the middle tree is called **staging area** or *index*, has many benefits described in the following use case:

> **Workflow:**
>
> A webpage developer is requested to change the contacts in the website, and they are in multiple pages... so he modifies all pages separately as staging, then he submits one commit to this patch of modifications.

Are you bothered?! git has a shorthand command to make both in one shot.

Ch.5: Git file states

- In the staging area, the content is stored along with its hash.

2 main states: tracked and untracked
untracked == not indexed
tracked: has 2 subtypes

unmodified	modified
no modifications occured in the WT.	modifications made without being staged.

> Under the hood:
> Once the file/dir is staged, the `.git` init an object with the hash of this blob/tree just for the sake tracking -and to be able to find diffs- till you explicitly commit the changes to be a snapshot.

Ch.7 Explore Git objects and trees:

ls	git ls-files
contents of WT	contents of the index. -s: shows permissions, hash of the file.

`git add <file_name>`

> can use wildcards in the <file_name>, like y* *.txt

`git cat-file - <file_full_hash>`:

- -t: type
- -s: size
- -p: content

git creates a 3rd object other than blob and tree: called '.commit' object that contains snapshot info(including the root directory of related files to this commit)

Ch.8: Basic Git ops:

`git status`:
`-s`: shorthand for the status, as it shows each file with a notation of its state, as follows
M: modified in WT, and not reflected to neither the index nor the repo
M: the modification is *added* to the index.

(contd.) Git arch.:

The commits are ordered in time as LinkedList, where each commit refers to its parent as the recent previous commit.. and the first commit has null parent so it is called the 'root commit'.
Branch: is the linear, logical ordering of group of commits.
Note: the sequence direction is in opposite to time direction, as you will experience with `git log`

`git commit`: removes the objects from the staging area.
`git diff`: by default, between staging and WT

- -staged: with each git command, is between the index and the repo

`git log`: can filter by file, owner, no. o' last commits

remove	rename
actually it addes to git commits	not recommended to be done using the OS, and the most safest way: `git mv`

Ch.9: Undoing things:

> Related commands are prescribed in the `git status` output*

to undo changes from the repo, we utilize the `HEAD`, similar idea to the recording device(e.g. walkman), in a process called *revision*
`git reset HEAD~n`, where n: number of returns onto the index, while `@{n}` no. of FFs
--hard: reflects directly on the WT, so be cautious!!
To show commits in front of the HEAD, use `git reflog`

Ch.9: Tags

is another object created in the repo, custom by the owner to mention a specific commit as version of an application (a milestone we can say)
`git tag -a vN.M -m "commit_msg"`

01_modern_data_stack_diagram.md

Ch.10, 11: branching

`git log --oneline --decorate --graph --all` #TODO: configure an alias as a shorthand. `alias <new_name>="command"`
At `git branch [<bran_name>] [option]`:
* refers to the HEAD, and the green-labeled branch is the currently used.
multiple branches result in non-linear development.
[option]-merged: shows merged branches onto the master
-r: if gives something, so this means you cloned a remote repo.
-M: rename the current, master, name
`git switch <bran_name>`, checkout is deprecated
`git merge <bran_name>`: with the HEAD to be on the master

Divergence history:

> occurs when you commit a change at master after the last commit of other branch, and so on.

3-way merge: in contrast with FF

```
omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master)
$ git merge testing
Auto-merging manip.py
CONFLICT (content): Merge conflict in manip.py
Automatic merge failed; fix conflicts and then commit the result.

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git diff
diff --cc manip.py
index a631f53,f55727a..0000000
--- a/manip.py
+++ b/manip.py
@@@ -23,4 -23,4 +23,8 @@@ for w in list(data["PrayerTimes"].keys(
                       ))

  #dd testing
- #master again
 -#1
++<<<<<<< HEAD
++#master again
++=======
++#1
++>>>>>>> testing

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git diff --staged
* Unmerged path manip.py

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git log --oneline --decorate --graph --all
* 5e3d0d8 (HEAD -> master) first divergence
| * 48e2bb2 (testing) 2nd testing commit
|/
* 5eb22b2 1st testing commit
* d324297 #dd
* 1c27721 (tag: v2.0) 2nd line
* 747ff20 yet init_commit

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git merge testing
fatal: You have not concluded your merge (MERGE_HEAD exists).
Please, commit your changes before you merge.

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)


omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master|MERGING)
$ git commit -m "VScode merge"
[master 0320fdd] VScode merge

omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (master)
$ git log --oneline --decorate --graph --all
*   0320fdd (HEAD -> master) VScode merge
|\
| * 48e2bb2 (testing) 2nd testing commit
* | 5e3d0d8 first divergence
|/
* 5eb22b2 1st testing commit
* d324297 #dd
* 1c27721 (tag: v2.0) 2nd line
* 747ff20 yet init_commit
```

`git rebase`: TODO!

Ch.12: Wrking w' remotes:

clone	push	pull/fetch
`git clone <url> <new_name>`		* to have the repo factually up to date. fetch: it doesn't reflect WT as a step of the workflow(to guarantee this update before merging). pull=fetch+merge with FF as normal

`git remote [option]`

-v: verboose

`disclaimer: you can't use -am till the first commit of the file itself by manually adding -the- files. As the indexing is the issuer for _this_ file to be tracked`

`git status: compares by branches, not the entire repo. git push origin fatal: The current branch feat has no upstream branch. To push the current branch and set the remote as upstream, use git push --set-upstream origin feat`

> What it actually does, firstly creates a branch with this name, then sync the commits of the local branch with the remote one created. and both branch names MUST match!

GitHub

Ch.13, 14: Intro to GitHub and basics_1

main == master

- It's not practical to create the entire repo from scratch on Github platform itself(a commit for each file added, modified, deleted), but for just one file: it's OK!
- Github features protection for certain branches, that doesn't exist in Git. As well as pull request.
- To create a branch, search for it.

Ch. 16: Authenticating to Push to GitHub:

HTTPS	SSH
using username/password. You need to enter the username, and password properly to successfully push your updates upstream to the hosted account on Github	using SSH key, for repos by other owners. Generating a new SSH key and adding it to the ssh-agent

```
omarn@LAPTOP-VIMT9GHF MINGW64 /h/Self_learning/Git/gitw (main)
$ git push -u origin_ssh main
To github.com:omarnegm2022/ddd.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'github.com:omarnegm2022/ddd.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

> Recall: `git log --all` to show fetched commits.

Final chapter: Forking the repos

if you want to contribute with others repo(s), this is a feature in Github as first step...
then you deal with your **forked** version.
As you open a contribution pull request to someone, you can close it yourself.

> Steps:
>
> 1. Press the 'fork' button on the top right of main repo page.
> 2. Complete the instructions normally.
> 3. Clone your own forked repo.
> 4. Make changes and commit them.
> 5. Push your final commit to your forked indeed.
> 6. Issue a pull request to the owner of original repo for merging your updates, if you think they are necessary.