1. GIT

Git is an open-source distributed version control system (Git tracks changes to source code). It is fast, smart, flexible, safe, and distributed. As problems, we have that any developer could access at any file (git has no access control) and is heavy, but we could we able to copy the latest commit or divide our project into multiple Gits projects. However, it is not useful for binary files like videos that are heavy.

2. Terminal Commands

Command	Definition
`pwd`	prints current directory
`cd [folderName]`	moves between directories
`ls`	list of files in a directory
`mkdir [folderName]`	creates a new folder
`touch [fileNAME]`	creates new empty file
`git --help`	gives the Git commands
`nano [fileNAME]`	allows edit a file from terminal

3. Installation

OS	Command
Debian	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	`https://git-scm.com/download/win`
Mac	`brew install git`

4. Basic Git Commands

Command	Definition
`git init`	initializes git (creates a repository)
`git add .`	we put all files in the launchpad called index
`git commit -a -m "[message]"`	creates snapshot that includes all files that are in the launchpad with a message, the -a option automatically stages all changes
`git log`	gives the history of the project: commitHash, author, date
`git log -[number]`	limit log for a specific number of commits
`git status`	gives the branch name and shows the files tracked (commit status)
`git status -s`	M=modified, A=new file added to staging area, ??=new file untracked by git
`git rm [fileName]`	remove file from tracked and project
`git rm --cached [fileName]`	remove file from project but continues on tracked files
`git --version`	gives the git version

5. Files Tracked by Git

Files Tracked	Definition
Committed	Unmodified changes from the last commit snapshot
Modified	Changes made to files since the last commit snapshot
Staged	Changes to be added into the next commit snapshot

6. Version Control

Gives a historical detail of your project. It is not a backup. It allows tracking a project's history. Furthermore, allows working with different versions (branch).

Command	Definition
`git branch [branchName]`	create a branch with the branchName
`git branch`	returns the list of local branches
`git branches -all`	returns all branches that exist in the server
`git switch [branchName]`	move between branches
`git checkout [branchName]`	move between branches
`git checkout -b [branchName]`	creates a new branch and move at this branch
`git checkout -b [commitHash]`	works into a commit without branch, this commit go to be deleted by garbage collector if a branch is not created
`git merge [branchName]`	to merge code between branches: go to the branch that needs the code of another branch and use the command merge. After this could be necessary resolve conflicts, because git does not the code that you need. Preserve history
`git rebase [branchName]`	insert commits of [branchName] downside of the current branch. After this could be necessary resolve conflicts. Refactor project history
`git tag [tagName]`	creates a tag
`git tag [tagName] [gitCode]`	annotated tag: reference a git code to use this tag or hash (lightweight tag)
`git log --graph --decorate --oneline`	show log using graph with colors
`git log --patch`	shows exactly what changes were introduced in the commit
`git log --grep [word] --oneline`	filter a log by word showing only one line for each founded word
`git log HEAD~5..HEAD^ --oneline`	filter 5 last commits of the current branch
`git log [currentBranch] [branch2] --online`	commits that are in the branch2 and not in our currentBranch
`git show [commitHash]`	detailed information about this commit
`git blame [fileName]`	gives the commits information that has a file

Error: the following untracked working tree files would be overwritten.... Implies that we have a file remove from one branch. To add this file

Command	Definition
`git add [fileName]`	add the fileName
`git stash`	store changes in a dirty working directory away

Client-server version control: The information is in a server, the developer copy the project to works. When the developer finished updating, he download again the project merge and fixed the conflicts, and finally made the commit in the server. Distributed version control: The project is cloned with commits (we use git), developers could synchronize changes with the team. First of all, it is necessary to make a pull before updating changes, fixing conflict, and making a push.

Command	Definition
`git clone [directory]`	allows to clone a repository that is called origin
`git fork [directory]`	allows to clone a repository and maintain relation with the third party repository that is called upstream
`git push`	allows to update developer changes
`git push -f`	force to copy exactly the branch given to the server. Delete information of the server
`git push origin [tagName]`	the tag go to be used as a release on Github
`git fetch`	copy the branch that has changed in the server to local repository, we can marge and fix conflict before made changes
`git pull`	allows to synchronize the repository (git fetch + git merge)

7. Creates a new repository on GitHub (code hosting provider)

Command	Definition
`touch README.md`	creates new empty file called README.md
`git init`	initializes git (creates a repository)
`git add .`	we put all files in the launchpad called index
`git commit -m “[message]”`	creates snapshot that includes all files that are in the launchpad with a message
`git remote add origin [link]`	connect with the GitHub repository
`git push -u origin main`	update the information on Github using the remote name “origin” and branch "main". The -u for every branch successfully pushed add upstream (tracking) reference. If we use a fork we could use “upstream” instead of “origin” depending if interact with our project or third party project

8. Git Three Rules

The current branch tracks new commits When you move to another commit, Git updates our working directory Unreachable objects are garbage collected

9. Four Git Areas

Git Area	Definition
Working Area	it is the location where we have our project
Repository	contains the entire history of the project (commits, trees and blobs)
Index (staging area)	the place where you put the files before a commit
Stash	temporary area. This area change only with stash commands. Help saving documents for use in different branches.

Command	Definition
`git diff`	compares working area with index
`git diff --cached`	compares index with repository
`git diff [commitHash1] [commitHash2]`	shows changes between two commits
`git diff --staged`	shows changes resume
`git add [. or fileName]`	copy to index
`git add --patch[fileName]`	applies different commits for one filename that was changed. Use To have more information: "n"to skip, "s"to obtain more hunks
`git commit -m "[message]"`	copy information to repository
`git checkout`	copy the information from repository to working area and index
`git checkout HEAD [filename]`	copy the repository file to index and working area
`git rm [fileName]`	remove file from working area and index
`git rm --cached[fileName]`	remove file only from index, it is opposite to command git add
`git mv [fileName] [newFileName]`	rename a filename
`git reset--soft [commitHash]`	go to a specific commits and copy from this commits: working area and index
`git reset--mixed [commitHash]`	go to a specific commits and copy from this commits: index
`git reset--hard [commitHash]`	go to a specific commits and copy from this commits: working area, index and repository in that moment the garbage collector delete information not used after –hard
`git reset --hard HEAD`	returns at the current state of the repository and delete any change in working area and index
`git reset HEAD`	copy only the index from repository
`git stash --include-untracked`	save files. The command –include-untracked specifies files that are not added. By default git stash ignore untracked files. After this command an git reset –hard HEAD is applied
`git stash list`	list of stash
`git stash apply`	copy information of the stage to the working area and index
`git stash clear`	delete the stash

10. Fixing Mistakes of History

Command	Definition
`git commit --amend`	git copies the latest commit and join with the new commit
`git rebase -I [commit]`	it allows to edit history of this commit onward, but shows in opposite order its commits use [wq] to write and quite. If there are a conflict: go to conflict, resolve and execute: git add and git rebase –continue
`git rebase --continue`	continue executing a git rebase
`git reflow [referenceName]`	gives the information about commits instead of they could be erased until garbage collector erase that information
`git filter-repo --path [fileName] --invert-paths`	delete all information and commit of the Filename
`git revert`	revert a commit creating a new commit that deletes that commits. Be careful reverting merges

11. Vocabulary

Word	Definition
`Untracked`	file is in the working area but not in the index or repository. Git does not what to do yet
`--patch`	command to analyze hunk by hunk could be used with add, checkout, stash, reset
`HEAD^^`	^ it refers to the parent of HEAD. If you use two ^^ you refers to the parent of the parent of the HEAD
`HEAD` $\sim 2 :=$ `HEAD^^`	if there are multiple commits you can specify the commit with HEAD ~ 2 ^ 2. The last 2 specify the commit

12. Configuration Levels (local, global, or system)

Command	Definition
`git config --local user.name "[name]"`	repository (local): repsoitoryName/.git/config
`git config --global user.name "[name]"`	user account (global): userDirectory/.gitconfig
`git config --systemuser.name "[name]"`	git installation (system): /usr/local/etc/gitconfig
`git config --list --show-origin`	shows all configurations
`git config --global user.name "[name]"`	config global name
`git config --global user.email "[email]"`	config global email
`git config user.name`	returns global name
`git config user.email`	returns global email
`git config --[level] --unset user.name`	remove a specific setting for a specific level
`git config --[level] --edit`	edit a specific level of config directly
`git config --[level] --remove-section user`	remove a section of config for a specific level
`git config --[level] core.editor "code --new-window --wait"`	configure visual studio code as default windows that go to be opened in a new window when it is necesssary
`git config --[level] core.editor "nano"`	configure nano as default editor
`git config --[level] -e`	review configuration in editor
`git config --[level] alias.[shortcut] [command]`	create a shortcut cut of a command
`git config --[level] alias.[shortcut] '[command1 command2]'`	create a shortcut of some commands
`./repositoryName/.gitattributes`	folders where can specify what type of files you have when git cannot detect satisfactory them

13. Configure Github Token (Github Token)

Command	Definition
`git config --global credential.helper cache`	after using your token, its goes to be stored

14. Use a Tool with Git (exiftool, example for images)

Command	Definition
`brew install exiftool`	install the tool in MAC
`mkdir repo`	create a folder called repo
`cd repo`	go to inside the repo folder
`git init`	initialize git
`echo "# Git Attributes" >> .gitattributes`	creates the file gitattributes with the title "# Git Attributes"
`git config --global diff.exif.textconv exiftool`	specify how git is going to utilize the exif strategy and what tools it is going to be leveraging

`nano .gitattributes`
`# Git Attributes`
`*.jpg diff=exif`
	open with nano editor the file .gitattributes
	
	define the attribute that go to use exiftool when performing diffs

`git diff`	now when an .jpg image is edited we could have more detail about changes that was made

15. Git Attributes to Show in a zip of the Release

Filter to change some words to another depending if we are on local repository or origin repository.

```
git config --local filter.[filterName].smudge 'sed "s/[ORIGIN_REPO_WORD]/[LOCAL_WORD]/"'
git config --local filter.[filterName].clean 'sed "s/[LOCAL_WORD]/[ORIGIN_REPO_WORD]/"'
```

`nano .gitattributes`
`# Git Attributes`
`.* filter=[filterName]`
	`edit the file according at your requeriments`
	
	`location where we go to apply the filter`

`.* export-ignore`	`ignore all attributes that start with . in the repository. This files does not appear in the zip of release`
`git push -u origin main`	`we made the push`
`git tag [tagName]`	`creates a tag`
`git push origin [tagName]`	`put specific tag to origin. This appears in the Github releases option`

16. Git Submodules

```
Two separate repositories are linked together within a project. Template iniside: repo/.gitmodules

[submodule "external/[repoName]"]
path = external/[repoName]
url = [repoUrl]
```

Command	Definition
git --version	greater than 2.09 to create submodules
cd [repositoryName]	go to repository
mkdir [submodelFolder]	creates a folder name for submodules
git submodule add [url] [submodelFolder]/[submodelName]	link submodule with external repository
git configuration --global status.submoduleSummary true	allows use git status command and obtain more details about commits in submodules
git config --global diff.submodule log	to obtain more detail in submodules summaries
git submodule init	initialize submodules
git submodule update	grant the content of the submodules
git submodule deinit external/[repoName]	removing a submodule from repository temporarily
git fetch	specify that we go to work with specific submodule and we can use normal commands. Firstly use cd [submodelFolder]
git submodule deinit external/[repoName] git rm external/[repoName] git commit -m “[commitMessage]”	remove submodule permanently from the repository

- Submodules are truly their own repository
- Do not automatically track to a branch but rather a specific commit
- Must be updated explicitly
- Can also contain other submodules
- Can be edited and updated just like a normal repository

17. Git Hooks (optimize aspects of workflow)

A script that is executed in response to a specific action within a Git repository. We have client-side hooks and server-side hooks. There are hooks inside /.git/hook.

18. Activate Client-Side Hooks

Command	Definition
git config --local core.hooksPath .githooks	configuration of folder .githooks to use hooks
mkdir .githooks	creates folder .githooks
mv .git/hooks/[fileName] .githooks	moves the hook that go to be used
chmod +x . githooks/[fileName]	gives execution permission

19. Activate Server-Side Hooks

Command	Definition
git clone --bare url ’[copyFolderName]’	clone -bare repository
chmod +x . githooks/[fileName]	gives execution permission at the file that represent your githook

20. Custom Git Commands

Command	Definition
mkdir git-scripts	creates a folder for our commands
cd git-scripts	go to the folder
nano git-[scriptFileName]	create file to edit our script. Should start with git-
echo “Test Script is Working”	information inside [scriptFileName] to observe that our script is working
chmod +x [scriptFileName]	gives execution permissions
echo ’export PATH=“[path_git -scripts]:$PATH”’ >> ∼/.bash_profile	add commands of the git-scripts folder to execute in your computer. Furthermore, add path in /etc/paths if you have a problem in MAC
source ∼/.bash_profile	update paths
git [scriptFileName]	if prints Test Script is Working. Then, all is correct

21. Git Bisect

A tool included with Git that enables you to specify a start and end of a commit. Manually or automatically determine the point of failure within that range of commits.

Command	Definition
git bisect start	initialize the bisect process
git bisect good [commitHash]	specify good commit
git bisect bad [commitHash]	specify bad commit
git bisect run npm test	automatically evaluate commits
git bisect reset	ending git bisect

22. Git Server

Command	Definition
git clone [/var/repositories/project.git]	example of clone using path
git clone [file:///var/repositories/project.git]	example of clone file explicity
git clone ssh://andres@gitserver:/var/repositories/project.git	example of clone using ssh
git clone andres@gitserver:/var/repositories/project.git	example of clone using scp
git clone https://gitserver/var/repositories/project.git	example of clone using https
git clone git://gitserver/var/repositories/project.git	example of clone using git protocol

23. Client

Command	Definition
sudo apt-get update && sudo apt-get -y upgrade	updates to install latest version of programs
sudo adduser [userName]	creates user in Linux
sudo usermod -aG sudo [userName]	creates a group for that user
su - [userName]	works as [userName]
git config --global user.name “[name]”	configures a global name
git config --global user.email “[email]”	configures a global email
mkdir ∼/.ssh	creates an ssh directory for that user
chmod 700 ∼/.ssh	gives access permissions only to the owner
ssh-keygen	generates an ssh key
cat .ssh/id_rsa.pub	shows the public generated

24. Server

Command	Definition
sudo apt-get update && sudo apt-get -y upgrade	updates to install latest version of programs
sudo adduser [userName]	creates user in Linux
sudo usermod -aG sudo [userName]	creates a group for that user
su - [userName]	works as [userName]
git config --global user.name “[name]”	configures a global name
git config --global user.email “[email]”	configures a global email
mkdir ∼ /.ssh	creates an ssh directory for that user
chmod 700 ∼ /.ssh	gives access permissions only to the owner
nano ∼ /.ssh/authorized_keys	we creates a file for people that could access at server and paste results of “shows the the public generated” command used in client
chmod 600 ∼ /.ssh/authorized_keys	gives permission for clients
mkdir [repoName].git	creates our repository folder
cd [repoName].git	go to the repository
git --bare init	start the git server

25. Client

Command	Definition
git clone git:[serverIp] /home/git/[repoName].git	The client could clone the repository from server

26. Server

Command	Definition
nano createUser.sh	creates commands that users need satisfies
nano populateUsers.sh	creates users calling before file
chmod 700 createUser.sh populateUsers.sh	allows execution permissions
sudo ./populateUsers.sh	creates users after obtain its ssh public key and paste in ∼ /.ssh/authorized_keys
which git-shell	shows where is installed git shell
sudo nano [git-shell-PATH]	add as valid login git-shell
sudo chsh git -s $(which git-shell)	updates where is git-shell. Now people could connect wih their name and not use git user

27. Client

Command	Definition
git clone git:[serverIp] /home/git/[repoName].git	The client could clone the repository from server

Note: It is possible says what user can or cannot made “$\sim /.ssh/authorized_keys$” adding before SSH_USER_KEY the correspondent code. For instance:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

28. Gitolite

A tool that assists in managing our Git server and it is necessary to use HTTP access together with APACHE.