**Kalpana Automation Task – I**

~Siddharth Subramanian

Q1) What is git and why is it used?

Git is a version control system. To note a version control system is a software that helps to keep track of the changes made to our code over time in a special database called a repository. It shows when, by whom and which file has been edited or modified.

This is useful in many ways such as backtracking mistakes and keeping track of our progress through the passage of time. It also supports many different people working on a single project. Hence it makes collaborating easier. Git comes under the distributed version control system where each person has a copy of the project and there is no centralized dependencies. So if the central server goes down the members would not be affected as each have an independent copy of the project to modify. GIT is commonly used mainly because of the following perks:

* Free to use
* Open source – Open and accessible to all
* Fast - Low latency
* Scalable – We can scale up and scale down whenever we want

Q2) Difference between GIT and GITHUB

|  |  |
| --- | --- |
| **GIT** | **GITHUB** |
| * GIT is a version control language which allows developers to track changes in their code as the project proceeds | * GITHUB is a web-hosting service to store our git repositories. In other words, GITHUB is the storage hub of git repositories |
| * U can use GIT without GITHUB | * GITHUB can’t be used without GIT |
| * It is locally based on your computer | * It is cloud based |
| * Git files are usually published to viewers who have a private set of people with whom they are collaborating | * GITHUB repositories can be accessed by anyone if it is made public or can be shared by limited individuals if made private |
| * Free and open source for all and at all times | * Free for basic operations, can upgrade to a paid premium plan for later on as well |
| * Git requires manual configuration of access control and other security measures | * GITHUB provides built in access control features, code encryption and a centralized security management |
|  |  |

Q3) What are the different types of version control?

There are mainly 4 types of version control systems.

* **Centralized Version Control system:** There is a central server which has a central repository which is accessible to everyone having the authorization with the client. This repository stores all the versions of the files, thereby all the new updates are delivered to everyone without any discrepancies. All the changes that are made within the files of the repository are to be committed to the centralized repository itself. This also causes some issues like lets say when the server is down then none of the users can access the repositories and none of the changes (if made) are saved. This can cause inconvenience to a certain extent. Example: Subversion, Perforce, ClearCase
* **Distributed Version Control system:** Here there is no centralized server, everyone who has the client authorization have a copy of the repository/project and everyone can make their individual changes and communicate their changes via snapshots. This also removes the dependence on a centralized repository and there is therefore no worries of the server going down as each person has a copy at all times. This also enables the users to work offline, offers stronger branching and merging features. While the drawback of communication always exists, collaborating over a cloud network like the OneDrive, GITHUB or Google drive can help in improving this. Ex: GIT, Memorial
* **Hybrid Version Control Systems:** It combines the best of both the Centralized and the Distributed version control systems, such as existence of a single centralized repository and controlled access from the Centralized version control system and the existence of copies of the original repository, offline working, and extensive branching and merging from the Distributed version control systems. This increases flexibility, improves collaboration issues (Attempts to fix mutual issues) and increased security. It is a model which tries to fix the issues of both the Centralized and the distributed models using each other. Examples: Subversions of GIT and perforce helix.
* **Local Version Control Systems:** It comes under the oldest forms of version control. It helps in recording and managing source code files over time. It works by storing changes in the form of patches, which are differences between files. Each patch only contains the updates implemented since the previous patch. Let us say , if you add a line of code to a file, the patch will only record that line, not the entire file. This way, you can save disk space and track the history of your files. But this is quite outdated due to many factors such as poor collaborations as it does not allow multiple collaborators in the same project. There is no backup or recovery in case of accidental deletion. The security patches are weak and there is no graphical interface, hence its tools like RCS, CVS and SCCS have been replaced by more advanced and reliable versions of centralized and distributed version control systems.

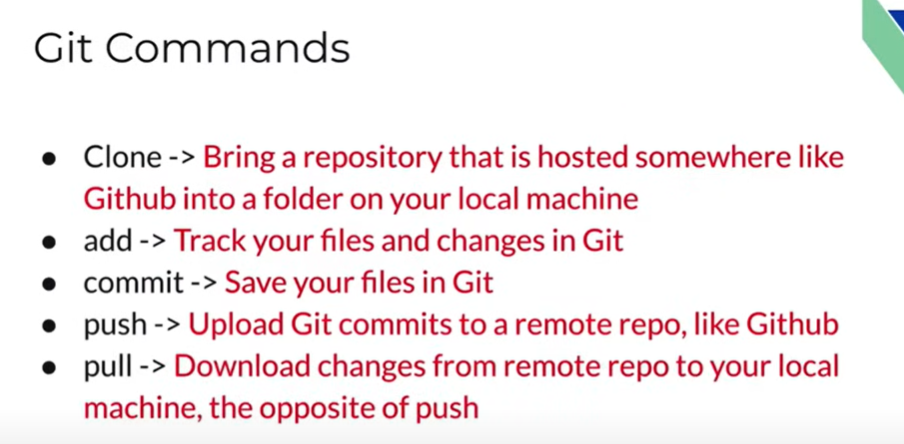
Q4) Basic GIT command

* **Git init:** This command creates a new local repository in the current directory. It initializes the .git folder that contains the metadata and configuration files for the repository.
* **Git clone**: This command copies an existing remote repository to the local system. It also sets up the remote origin and the master branch for the local repository.
* **Git add:** This command adds one or more files to the staging area, which is a temporary area where changes are prepared before committing. You can use git add. to add all the files in the current directory, or
* **Git commit**: This command records the changes in the staging area to the local repository. It also requires a message that describes the changes. You can use git commit -m "message” to commit with a message, or git commit -a to commit all the changes since the last commit.
* **Git push**: This command sends the changes in the local repository to the remote repository. It also updates the remote branches and references. You can use git push origin master to push the master branch to the origin remote, or git push --all origin to push all the branches.
* **Git pull**: This command fetches the changes from the remote repository and merges them with the local branch. It is equivalent to git fetch followed by git merge. You can use git pull origin master to pull the master branch from the origin remote, or git pull --rebase to rebase the local branch on top of the remote branch.
* **Git status**: This command shows the current state of the working directory and the staging area. It also displays the files that are modified, staged, untracked, or conflicted.
* **Git diff**: This command shows the differences between the files in the working directory and the staging area, or between the staging area and the local repository. You can use git diff to see the changes that are not staged, or git diff --staged to see the changes that are staged.
* **Git log**: This command shows the history of the commits in the local repository. It also displays the author, date, message, and hash of each commit. You can use git log --oneline to see a concise summary of the commits, or git log --graph to see a graphical representation of the commit tree.
* **Git branch**: This command creates, lists, renames, or deletes branches in the local repository. A branch is a pointer to a specific commit that represents a parallel line of development. You can use git branch <branchname> to create a new branch, or git branch -d <branchname> to delete a branch.
* **Git merge**: This command combines the changes from one branch into another branch. It also creates a new commit that represents the merge. You can use git merge <branchname> to merge a branch into the current branch, or git merge --abort to cancel a merge in progress.
* **Git reset**: This command resets the current branch to a different commit. It also updates the staging area and the working directory to match the commit. You can use git reset --hard <commit> to reset the branch, staging area, and working directory to a commit, or git reset --soft <commit> to reset only the branch to a commit.
* **Git stash:** This command saves the changes in the working directory and the staging area to a temporary area called the stash. It also restores the working directory and the staging area to a clean state. You can use git stash to stash the changes, or git stash pop to apply the changes from the stash and remove them from the stash.
* **Git tag**: This command creates, lists, or deletes tags in the local repository. A tag is a name that points to a specific commit that represents a milestone or a release. You can use git tag <tagname> to create a lightweight tag, or git tag -a <tagname> -m "message" to create an annotated tag with a message.
* **Git remote**: This command manages the remote repositories that are connected to the local repository. A remote repository is a repository that is hosted on a different machine or a server. You can use git remote add <name> <url> to add a new remote, or git remote -v to list the existing remotes.
* **Git fetch**: This command downloads the changes from the remote repository and updates the remote branches and references. It does not merge the changes with the local branch. You can use git fetch origin to fetch the changes from the origin remote, or git fetch --all to fetch the changes from all the remotes.
* **Git config**: This command configures the settings and preferences for Git. It also stores the user name, email, editor, and other options. You can use git config --global user.name "name" to set the user name for all the repositories, or git config --local user.name "name" to set the user name for the current repository.
* **Git help**: This command displays the help information for Git commands. It also shows the usage, options, and examples for each command. You can use git help <command> to see the help for a specific command, or git help -a to see a list of all the commands.

Q5) Using GIT and GITHUB seamlessly from your local machine

Works on Command line interface, it is case sensitive

GIT Commands:



Readme.md - file that contains all the necessary info of the project/repository. Usually a text file

Each commit has a unique address mentioned

While importing to our code editor, use the HTTPS link not the SSH one. Unless we have an SSH key setup we cannot use SSH.

LS - short for la-ls. It is used to list the folders and files within the folders including hidden files.

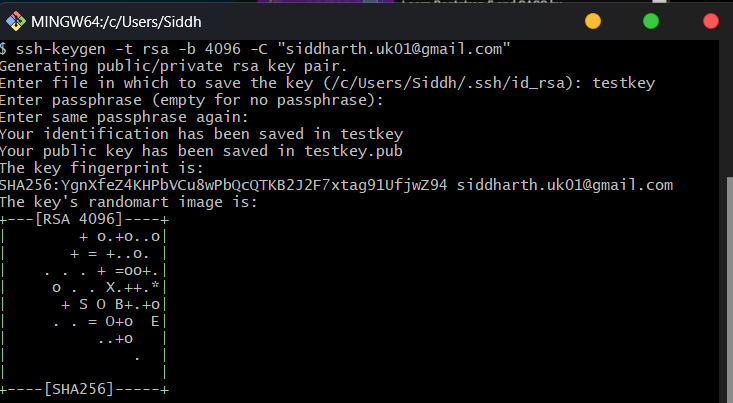
git status - command used to show history and shows the files that have yet to be committed.

When you get an untracked file, it shows that the particular file hasn’t been updated in git

To commit an untracked file, you need to need to use "git add . " this will track both the untracked files in the repository and update the files that have been updated in the mean time.

If we want to commit a particular file then use git add <File name>

Connecting our machine to GITHUB is done through SSH keys



This can be run on cmd prompt, GIT and on VSCode.

The public key given by <testkey>.pub is a proof that this public key exists because of private key

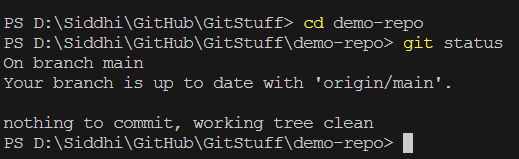
To copy anything to our clipboard, we use pbcopy < ~/testkey.

Just use ~ to copy previous text to clipboard

Use git push origin main to save file to repository in GITHUB. It requires authentication then you can upload

**To save file into the repository in GITHUB we do**

1. **git add .**
2. **git status**
3. **git commit -m "<Title>" -m "<Description>"**
4. **git push origin main**
5. **Refresh GITHUB to access the files**



Cd demo-repo gives changes the directory or leads to git to the repository demo-repo

Git status is used to check if the repository we're in is upto date or any files are pending that require pushing and committing.

Git remote is used to access a repository that is not on this computer, but access it locally through the command