What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

* Tracking code changes
* Tracking who made changes
* Coding collaboration

What does Git do?

* Manage projects with **Repositories**
* **Clone** a project to work on a local copy
* Control and track changes with **Staging** and **Committing**
* **Branch** and **Merge** to allow for work on different parts and versions of a project
* **Pull** the latest version of the project to a local copy
* **Push** local updates to the main project

Working with Git

* Initialize Git on a folder, making it a **Repository**
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is changed, added or deleted, it is considered **modified**
* You select the modified files you want to **Stage**
* The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
* Git allows you to see the full history of every commit.
* You can revert back to any previous commit.
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

git config --global user.name ""

git config --global user.email ""

**md gitpractice**

**cd gitpractice**

**d\gitpractice>git init**

**d\gitpractice> excho > abc.txt**

**d\gitpractice> excho “ copy some text “> abc.txt**

**d\gitpractice> type abc.txt**

**D:\Git\gitpractice>git status**

**On branch master**

**No commits yet**

**Untracked files:**

**(use "git add <file>..." to include in what will be committed)**

**abc.txt**

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

Files in your Git repository folder can be in one of 2 states:

* Tracked - files that Git knows about and are added to the repository
* Untracked - files that are in your working directory, but not added to the repository

 When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

**Staging environment:**

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

**Staged** files are files that are ready to be **committed** to the repository you are working on.

* + **git add filename**
  + **git add all**
  + **git add .**

**will make the files to staging area. Can check with**

* **git status**

Git Commit

* Since we have finished our work, we are ready move from stage to commit for our repo.
* Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.
* When we commit, we should **always** include a **message**.

git commit -m "First release of Hello World!"

Git Commit without Stage (-a)

git commit -a -m "Updated index.html with a new line"

Git Commit Log

To view the history of commits for a repository, you can use the log command:

git log

Git Help

If you are having trouble remembering commands or options for commands, you can use Git help.

There are a couple of different ways you can use the help command in command line:

* git *command* -help -  See all the available options for the specific command
* git help --all -  See all possible commands

Working with Git Branches

In Git, a branch is a new/separate version of the main repository.

Let's say you have a large project, and you need to update the design on it.

How would that work without and with Git:

Without Git:

* Make copies of all the relevant files to avoid impacting the live version
* Start working with the design and find that code depend on code in other files, that also need to be changed!
* Make copies of the dependant files as well. Making sure that every file dependency references the correct file name
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Save all your files, making a note of the names of the copies you were working on
* Work on the unrelated error and update the code to fix it
* Go back to the design, and finish the work there
* Copy the code or rename the files, so the updated design is on the live version
* (2 weeks later, you realize that the unrelated error was not fixed in the new design version because you copied the files before the fix)

With Git:

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix
* Fix the unrelated error and merge the small-error-fix branch with the main branch
* You go back to the new-design branch, and finish the work there
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

D:\Git\gitpractice>**git init**

Reinitialized existing Git repository in D:/Git/gitpractice/.git/

D:\Git\gitpractice>**git add .**

D:\Git\gitpractice>**git commit -m "first commit"**

[master (root-commit) 33b3753] first commit

1 file changed, 1 insertion(+)

create mode 100644 abc.txt

D:\Git\gitpractice>**git branch A**

D:\Git\gitpractice>**git checkout A**

Switched to branch 'A'

D:\Git\gitpractice>**git checkout master**

Switched to branch 'master'

D:\Git\gitpractice>**git branch B**

D:\Git\gitpractice>**git checkout B**

Switched to branch 'B'

D:\Git\gitpractice>**git checkout master**

Switched to branch 'master'

D:\Git\gitpractice>

Git status can be used always to check status of each branch.

**Emergency branch**

D:\Git\gitpractice>**git checkout -b Cemergencybranch**

Switched to a new branch 'Cemergencybranch'

Do changes

D:\Git\gitpractice>**git checkout master**

Switched to master

D:\Git\gitpractice>**git merge** **Cemergencybranch**

**Delete a branch:**

D:\Git\gitpractice>**git branch –d Cemergencyranch**

**To display all branches:**

D:\Git\gitpractice>**git branch**

A

B

hello-world-images

\* master

**Git log: (to exit q or z)**

D:\Git\gitpractice>**git log**

commit 45b1857dd87b10e3af5cc755afb52ef943b07cb1 (HEAD -> master)

Author: arunavan <arunajava567@gmail.com>

Date: Fri Mar 17 14:46:34 2023 +0530

commit 2

commit b16ec7689018085d8c7df742679ea7a422d06f19 (hello-world-images)

Author: arunavan <arunajava567@gmail.com>

Date: Fri Mar 17 14:43:11 2023 +0530

branch commit

commit 33b3753025194ebbc3a62a0d4407314a5f0f78d8 (B, A)

Author: arunavan <arunajava567@gmail.com>

Date: Fri Mar 17 14:23:02 2023 +0530

first commit

D:\Git\gitpractice>

Rebase Branches

The Git rebase command is a branch merge command, but the difference is that it modifies the order of commits.

The Git merge command tries to put the commits from other branches on top of the HEAD of the current local branch. For example, your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then git merge will convert the current local branch to something like A−>B−>C−>D−>X−>Y

The Git rebase command tries to find out the common ancestor between the current local branch and the merge branch. It then pushes the commits to the local branch by modifying the order of commits in the current local branch. For example, if your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then Git rebase will convert the current local branch to something like A−>B−>X−>Y−>C−>D

**Stash:**

you want to switch branches for customer escalation, but you don’t want to commit what you’ve been working on yet; so you’ll stash the changes. To push a new stash onto your stack, run the **git stash** command.

**git stash,**

**git stash list**

**git stash pop**

D:> git checkout –b newbranch

D:\>echo “hello”> abc.txt

D:\>git add .

D:\>git stash

D:\>echo “hye”> abcd.txt

D:\>git add .

D:\>git stash

D:\>git status

D:\>git checkout master

D:\>git status

D:\>..... do any work with master

D:\>git checkout newbranch

D:\>git stash list

D:\>git stash pop

...do necessary changes

D:\>git commit –m “messge”