1. GIT

* **What is Git**
  + Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people
  + Git does not necessarily rely on a central server to store all the versions of a project’s files.
  + Git helps you keep track of the changes you make to your code.
  + Git also helps you synchronise code between multiple people.
* **What is a Repository**
  + A repository a.k.a. repo is nothing but a collection of source code.
  + If you consider a file in your Working Directory, it can be in three possible states.
    - **Staged**
      * Which means the files with the updated changes are marked to be committed to the local repository but not yet committed.
    - **Modified**
      * Which means the files with the updated changes are not yet stored in the local repository
    - **Committed**
      * Which means that the changes you made to your file are safely stored in the local repository.
* **Git command**
  + **git add**
    - git add is a command used to add a file that is in the working directory to the staging area
  + **git commit**
    - git commit is a command used to add all files that are staged to the local repository
  + **git push**
    - git push is a command used to add all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository
  + **git fetch**
    - git fetch is a command used to get files from the remote repository to the local repository but not into the working directory.
  + **git merge**
    - git merge is a command used to get the files from the local repository into the working directory.
  + **git pull**
    - git pull is command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge .
* **GitHub Account and Configuration** 
  + Step 0: Make a GitHub Account. Duh.

<https://github.com/join>

* + Step 1: Make sure you have Git installed on you machine. If you are running Linux(deb), enter the following in the terminal
    - $ **sudo apt install git-all**
  + Step 2: Tell Git who you are. (Configuration)
    - **git config --global user.name "YOUR\_USERNAME"**
    - **git config --global user.email "im\_satoshi@musk.com"**
    - **git config --global --list # To check the info you just provided**
  + create new file and changes on file push this file in git repo

* + Git status
  + Git add .
  + Git commit -a “”
  + Uncommit Changes you just made to your Git Repo
* **Add a remote origin and Push:**
  + git remote add origin remote\_repository\_URL
  + git remote -v
* **Push code in master branch** 
  + git push
    - Code push local code to live repo in github
    - git push command pushes the changes in your local repository up to the remote repository
    - use origin
  + See the Changes you made to your file:
    - **git diff**
  + Revert back to the last committed version to the Git Repo
    - git checkout .
    - git checkout -- <filename>
  + View Commit History
    - git log
    - $ git add .
    - $ git status # Lists all new or modified files to be committed
    - $ git commit -m "Second commit"
    - $ git push -u origin master
    - Adding Files for Git to Track
      * we do not have any files for Git to track. We need to add files specifically to Git order to tell Git to track them. We add files using add
    - Removing Files
      * running a simple git rm will not only remove it from Git, but will also remove it from your local file system as well
  + Step 5 : That’s all well and good… But How do I download and work on other repositories on GitHub?
    - Cloning a Git Repo
      * **git clone remote\_repository\_URL**
    - Now you can work on the files you want and commit to changes locally.
    - So imagine you and your friend are collaborating on a project. You both are working on the same project files. Each time you make some changes and push it into the master repo, your friend has to pull the changes that you pushed into the git repo. Meaning to make sure you’re working on the latest version of the git repo each time you start working, a git pull command is the way to go.
      * **git pull origin master**
    - In the simplest terms, git fetch followed by a git merge equals a git pull. But then why do these exist
  + **Gitignore**
    - .gitignore tells git which files (or patterns) it should ignore
* **Make GIT repository**
  + Create account or login accont
  + Create new repository github account near profile plus to new repository option it click
  + select public and set the name of Repo name
  + Submit as create repository
* **Clone GIT repository**
  + When you create a repository on GitHub.com, it exists as a remote repository. You can clone your repository to create a local copy on your computer and sync between the two locations
  + On GitHub.com, navigate to the main page of the repository.
  + Above the list of files, click Code.
  + Copy the URL for the repository.
  + To clone the repository using HTTPS, under "HTTPS", click
  + Change the current working directory
  + Run git command
    - git clone
    - $ git clone <https://github.com/YOUR-USERNAME/YOUR-REPOSITORY>
* **Add global conguration like name, email**
  + To set your global commit name and email address run the git config command with the --global option:
  + git config --global user.name "YOUR\_USERNAME"
  + git config --global user.email "im\_satoshi@musk.com"
  + git config --global --list # To check the info you just provided
* **Add files to GIT repository**
  + In the command line, navigate to the root directory of your project.
  + Initialize the local directory as a Git repository.
    - git init -b main
  + Stage and commit all the files in your project.
    - git add .
    - git commit -m "initial commit"
    - git add . && git commit -m "initial commit"
  + Push the changes in your local repository to GitHub.com
    - git push origin main
    - Pushes the changes in your local repository up to the remote repository you specified as the origin
    - You can upload an existing file to a repository on GitHub.com using the command line.
* **Pull GIT repository**
  + the git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content.
  + The git pull command is actually a combination of two other commands, git fetch followed by git merge
  + The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit.
  + **git pull <remote>**
  + Git rebase
  + Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use git rebase
  + The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits.
* **Check file difference before commit object**
  + You want to see what changes you're going to commit compared to your last commit
  + These are common issues and tasks when working with a version control system. Fortunately, you can verify all this using the Git diff command.
  + git diff lists out the changes between your current working directory and your staging area.
  + You can also run the git diff <branch\_name1> <branch\_name2> command to compare the changes from the first branch with changes from the second branch
  + git diff HEAD <file\_name>
  + git diff <file\_name>
  + git diff --staged <file\_name>
  + git diff <branch\_name1> <branch\_name2> <file\_name>
  + git diff <commit\_hash> <commit\_hash> <file\_name>
* **Commit files to GIT repository**
  + Before we make a commit, we must tell Git what files we want to commit his is called staging and uses the add command.
  + That's where Git's add command comes in. We add files to a staging area, and then we commit what has been staged.
  + the deletion of a file must be tracked in Git's history, so deleted files must also be staged and then committed.
  + Commit files
    - Enter this command:
      * git commit -m "Message that describes what this change does"
    - Check the status again by running this command:
      * git status
    - If all changes have been committed, and there are no untracked files, it should say: nothing to commit, working tree clean
    - View a List of Commits
      * To see a simplified list of commits, run this command:
        + git log –oneline
      * To see a list of commits with more detail (such who made the commit and when), run this command
        + git log
* **Check git status**
  + The git status command displays the state of the working directory and the staging area.
  + List which files are staged, unstaged, and untracked.
  + The git status command is a relatively straightforward command.
  + Command
    - git status
* **Undo changes from modified file**
  + Finding what is lost: Reviewing old commits
  + How to undo a commit with git checkout
  + You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project
  + git checkout a1e8fb5
  + This assumes that you're developing on the default main branch. Once you’re back in the main branch, you can use either git revert or git reset to undo any undesired changes
  + Undoing a committed
  + How to undo a commit with git reset
  + git reset is an extensive command with multiple uses and functions. If we invoke git reset --hard a1e8fb5 the commit history is reset to that specified commit
  + Undoing the last commit
    - git commit –amend
  + A reset should be used when undoing changes to a private branch. This safely isolates the removal of commits from other branches that may be in use by other developers. Problems arise when a reset is executed on a shared branch and that branch is then pushed remotely with git push. Git will block the push in this scenario complaining that the branch being pushed is out of date from the remote branch as it is missing commits.
  + The preferred method of undoing shared history is git revert. A revert is safer than a reset because it will not remove any commits from a shared history. A revert will retain the commits you want to undo and create a new commit that inverts the undesired commit. This method is safer for shared remote collaboration because a remote developer can then pull the branch and receive the new revert commit which undoes the undesired commit.
  + git reset
* **Reset modified Files**
  + git checkout, git reset, and git restore are commands that can help you revert to a previous version not just of your codebase, but of individual files,
  + As software for version control, Git makes it predictably easy to roll back changes
  + The Git command to do this is, somewhat unintuitively, named checkout. You might’ve used checkout before to switch branches
    - git checkout
  + How Do I Change What I’m Staging
    - The other type of undo involves your staging area. This contains the changes you’ll commit next to the repository. If you’ve staged a version of a file and later decide you don’t want to include it in the next commit, you’ll want to unstage it first
      * You can use the git reset command to unstage a staged file:
    - git restore
      * To undo a change to the repository
      * git restore
      * git restore --staged path/to/file
      * git reset HEAD path/to/file.
* **Reset committed objects before push**
  + The git reset command is a complex and versatile tool for undoing changes
  + git reset b
  + the default invocation of git reset has implicit arguments of --mixed and HEAD
  + –hard, --mixed,--soft
  + –mixed
    - This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory
  + –soft
    - When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched. This behavior can be hard to clearly demonstrate. Let's continue with our demo repo and prepare it for a soft reset.
  + git reset ＜file＞
  + git reset HEAD~2
    - The git reset HEAD~2 command moves the current branch backward by two commits
  + git reset
  + git reset --hard 802fe3b
* **Push committed files to GIT repository**
  + The git push command is used to transfer or push the commit, which is made on a local branch in your computer to a remote repository like GitHub
  + Add the file to the new local repository
    - git add .
  + Commit the files staged in your local repository by writing a commit message
    - git commit -m "Add existing project files to Git"\
  + Push the code in your local repository to GitHub
    - git push -u origin master
    - the origin is your default remote repository name and '-u' flag is upstream, which is equivalent to '-set-upstream.'
* **Check GIT logs**
  + you should already know the basic git log command for displaying commits
  + formatting how each commit is displayed, and filtering which commits are included in the output
  + Formatting Log Output
    - First, this article will take a look at the many ways in which git log’s output can be formatted
    - Oneline
      * The --oneline flag condenses each commit to a single line
      * git log –oneline
    - Decorating
      * Many times it’s useful to know which branch or tag each commit is associated with
      * git log --oneline –decorate
    - Diffs
      * the git log command includes many options for displaying diffs with each commit.
    - Shortlog
      * git shortlog
    - Graphs
      * git log –graph
      * The --graph option draws an ASCII graph representing the branch structure of the commit history
      * G--oneline –decorate
  + **Filtering the Commit History**
    - **By Amount**
      * The most basic filtering option for git log is to limit the number of commits that are displayed
      * You can limit git log’s output by including the - option. For example, the following command will display only the 3 most recent commits.
      * git log -3
    - **By Date**
      * git log --after="2014-7-1"
      * git log --after="yesterday"
      * git log --after="2014-7-1" --before="2014-7-4"
    - **By Author**
      * When you’re only looking for commits created by a particular user, use the --author flag
        + git log --author="John"
    - **By Message**
      * To filter commits by their commit message, use the --grep flag.
      * git log --grep="JRA-224:"
    - **By File**
      * git log -- foo.py bar.py
    - **By Content**
      * git log -S"Hello, World!"
    - git log –merges
* **Show particular committed object**
  + git-show is a command line utility that is used to view expanded details on Git objects such as blobs, trees, tags, and commits.
  + Git-show options
  + --abbrev-commit
    - This option shortens the length of output commit IDs. Commit IDs are 40 characters long and can be hard to view on narrow terminal screens. This option combined with
    - --pretty=oneline can produce a highly succinct git log output
  + -no-abbrev-commit
    - Always Show the full 40 character commit ID. This will ignore --abbrev-commit and any other options that abbreviate commit IDs like the --oneline forma
  + –oneline
    - This is a shortcut for using the expanded command
  + git show commitA...commitD
* **Stash the changes and how to apply changes back from stash**
  + Stash the changes and how to apply changes back from stash
    - Commands like git stash and git stash pop are used to Shelve (stash) and restore changes in our working directory.
    - Stashing the Changes in Working Directory we can use the git stash command to shelve our changes temporarily.
    - git stash
    - if do a git status again, we will see that our working directory is clean
  + Restoring Stashed Changes and Finding the Hash
    - Restoring Stashed Changes into the Working Directory
    - We can bring the stashed changes back into our working directory like this
    - git stash pop
  + Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:
    - git stash apply
  + Viewing stash diffs
    - git stash show
  + Creating a branch from your stashed
    - If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to
    - git stash branch add-stylesheet stash@{1}
  + Cleaning up your stash
    - git stash drop stash@{1}
  + Or you can delete all of your stashes with:
    - git stash clear
* **GIT multiple branches**
  + There are times where you might be working from a particular git branch and need to quickly jump over to a different branch to do some urgent work
  + Typically you would need to first git stash anything you were working on (as it’s unlikely to be in a state where it can be committed), and then you’d have to leave your current branch to create a new branch from master and thus begin working on your new urgent task.
  + Save the code you're working on. You could use git stash --all to save the changes and any new files for all
  + Switch to the other branch.
    - git checkout -b foo\_contents
  + Git worktree
    - Git worktree is one of the underrated features of Git that helps you to create a copy of the working repo so that you can work on it too.
    - Worktree unlike cloning is a soft copy of the repo so that it is lightweight and fast.
    - git worktree add <path> <branch\_name>
  + Listing worktrees
    - git worktree list
  + Removing worktrees
    - git worktree remove <worktree-name>
* **Create local GIT branch**
  + How to Create Branches in Git
    - In essence, there are two methods in Git for creating branches.
    - You can use a single command to create the branch and switch to it. Or you can create the branch first using one command and then switch to it later using another command when you wish to work with it
    - create the branch and switch to it
      * git checkout -b <branch-name>
    - create the branch
      * git branch <branch-name>
    - switch
      * git checkout <branch-name>
    - after code push into branch
    - git push -u origin <branch-name>
* **Push local GIT branch and make it remote branch**
  + I’ll show you how to push a local git branch to a remote server.
  + How to Push the Main Branch to Remote
    - If you want to push the main branch to remote, it’s possible you’re pushing for the first time. Before you attempt to push to remote, make sure you’ve executed these commands
    - git init for initializing a local repository
    - git add . to add all your files that the local repository
    - git commit -m ‘commit message’ to save the changes you made to those files
    - To finally push the repo, run git push -u origin <branch-name>
    - git push origin feature
  + Push Branch to Another repository
    - In order to push a branch to another repository
    - “git push” command
    - git push <remote> <branch>
    - Awesome, you pushed your branch to another remote repository!
* **Move between GIT branches**
  + How to create a new branch in Git
  + To create a new branch in Git, you use the git checkout command and pass the -b flag with a name.
  + This will create a new branch off of the current branch. The new branch's history will start at the current place of the branch you "branched off of."
  + How to switch to an existing branch in Git
  + To switch to an existing branch, you can use git checkout again (without the -b flag) and pass the name of the branch you want to switch to:
  + git checkout
  + You can use the git switch - command to undo any changes you make and return to your previous branch
  + If you instead want to keep your changes and continue from here, you can use git switch -c <new-branch-name> to create a new branch from this point.
  + git checkout <new-branch-name>
  + git checkout branch
* **Cherry-pick objects from one branch to another branch** 
  + git cherry-pick is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD
  + Cherry picking is the act of picking a commit from a branch and applying it to another.
  + git cherry-pick can be useful for undoing changes.
  + When to use git cherry pick
    - git cherry-pick is a useful tool but not always a best practice. Cherry picking can cause duplicate commits and many scenarios where cherry picking would work, traditional merges are preferred instead
  + How to use git cherry pick
    - git cherry-pick usage is straight forward and can be executed like:
      * git cherry-pick commitSha
      * commitSha is a commit reference
      * You can find a commit reference by using git log
      * Git log
    - git checkout main
    - git cherry-pick f
* **Continue cherry-pick and abort cherry-pick process**
  + git cherry pick can also be passed some execution options
    - **--edit**
      * Passing the -edit option will cause git to prompt for a commit message before applying the cherry-pick operation
    - **--no-commit**
      * The --no-commit option will execute the cherry pick but instead of making a new commit it will move the contents of the target commit into the working directory of the current branch.
    - **--signoff**
      * The --signoff option will add a 'signoff' signature line to the end of the cherry-pick commit message
* **Merge changes from one branch to another branch**
  + The "merge" command is used to integrate changes from another branch
  + While Git can perform most integrations automatically, some changes will result in conflicts that have to be solved by the user.
  + Important Options
    - **--no-ff**
      * Creates a merge commit even when a fast-forward would be possible
    - **--squash**
      * Combines all integrated changes into a single commit, instead of preserving them as individual commits.
    - **--abort**
      * When a conflict occurs, this option can be used to abort the merge and restore the project's state as it was before starting the merge.
  + Before using "git merge", make sure the correct local branch is checked out.
  + Then, to perform the merge, simply specify which branch's commits you want to integrate:
  + git checkout develop
  + git merge feature/login
* **Rebase from one branch to another branch**
  + The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another
  + As an alternative to merging, you can rebase the feature branch onto main branch using the following commands:
  + git checkout feature
  + git rebase main
  + Interactive Rebasing
    - Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch
    - To begin an interactive rebasing session, pass the i option to the git rebase command:
    - git checkout feature
    - git rebase -i main
    - git checkout feature git rebase -i HEAD~3
      * Specifying HEAD~3 as the new base, you’re not actually moving the branch—you’re just interactively re-writing the 3 commits that follow it. Note that this will not incorporate upstream changes into the feature branch.
* **Clone GIT repository with specific branch**
  + Now let's clone a specific branch from our demo repository. There are two ways to clone a specific branch. You can either:
  + Clone the repository, fetch all branches, and checkout to a specific branch immediately
  + Clone the repository and fetch only a single branch.
  + git clone --branch <branchname> <remote-repo-url>
  + git clone -b <branchname> <remote-repo-url>
  + With this, you fetch all the branches in the repository, checkout to the one you specified
  + git clone -b passwordless-auth [git@github.com:BolajiAyodeji/nextjs-blog.git](mailto:git@github.com:BolajiAyodeji/nextjs-blog.git)
* Reference Links
  + 1. https://www.sitepoint.com/git-for-beginners/
  + 2. <https://www.freecodecamp.org/news/learn-the-basics-of-git-in-under-10-minutes-da548267cc91/>
  + 3. <https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet>
  + 4. http://internal-docs.fads6.areads.com/index.php?id=91 [Search command by GIT]
    - Password and UserName
    - UserName
      * internal
    - Password
      * internal@fads6docs

Thanks you ..