Content

What is Git?

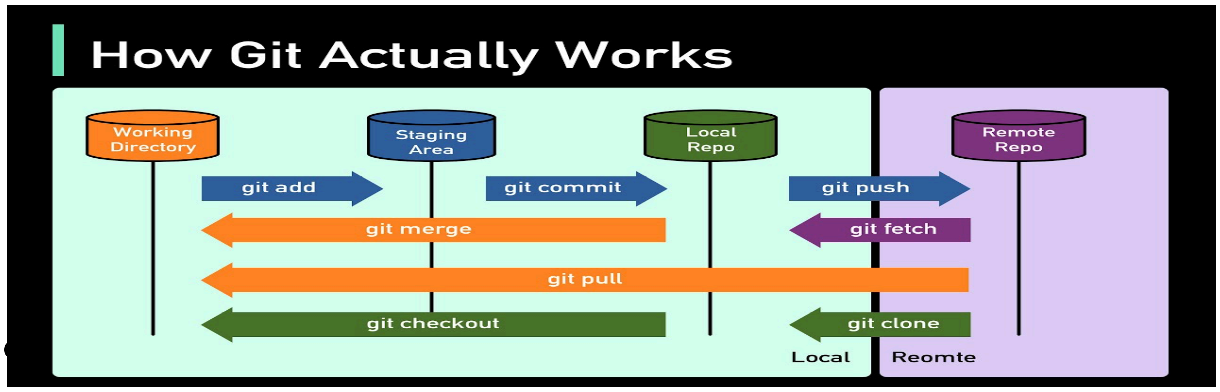
Git is like a time machine for your code. It tracks changes you make to your files, so you can undo mistakes, collaborate with others, and keep a history of your project's progress.

Understanding with a common example

Imagine you have a folder on your computer where you're working on a project, like writing an essay. Every time you make a change to the essay, you save it. Now, let's say you want to try something different, but you're worried you might mess up the essay. What if you could take a snapshot of the essay every time you make a change, so you could always go back to an earlier version if needed? That's kind of (not exactly) what Git does, but for computer files.

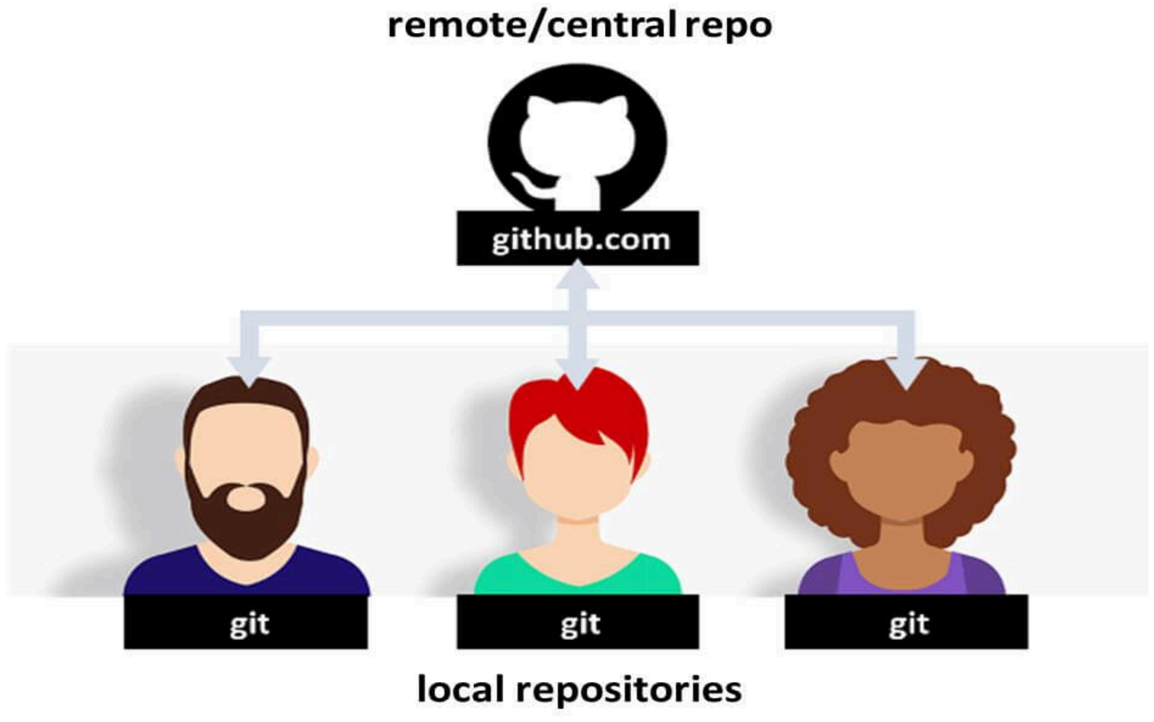
Let's Understand in Detail

Git is a distributed version control system that tracks changes to files and directories. It allows multiple users to collaborate on projects simultaneously. When you initialize a Git repository in a directory, Git starts tracking changes to the files in that directory. Each time you make a change to a file, you can "commit" those changes to the repository, creating a snapshot of the file's state at that point in time. These snapshots form a version history, allowing you to review or revert to previous versions of the files.



Remote vs. Local Code

When working with Git, you'll often come across the terms "remote" and "local" code. Let's break down what they mean in simpler terms.



Local Code

Local code is the code that lives on your own computer. It's where you write, edit, and test your projects. Imagine your local code as your personal workspace where you build and shape your creations.

Example: You're working on a cool website project called "MyWebsite" on your computer. All the files and folders related to this project, like HTML, CSS, and JavaScript files, reside in a folder on your desktop.

Remote Code

Remote code, on the other hand, refers to code that exists on a server or another computer, usually accessed over the internet. This could be code hosted on platforms like GitHub, Bitbucket, or GitLab. Remote repositories serve as centralized hubs where multiple people can collaborate on the same project.

Example: You decide to share your "MyWebsite" project with your friend who lives in another city. You upload your project code to a website like GitHub or Bitbucket, making it accessible to your friend over the internet. Now, both you and your friend can work on the same project, even though you're in different locations.

Branches

Imagine you're working on a group project with your classmates, each contributing different parts of a program. Creation: Branches are like creating separate copies of your project's code. You start a new copy (branch) to work on a specific feature or fix without changing the main version/code.

Isolation: Whatever changes you make in your branch stay there until you're ready to share them. This means you can experiment with new code without affecting the main project.

Collaboration: Your classmates also have their own copies of the project. You can all work on different parts of the program at the same time, like different functions or modules.

Merging: When you're done with your changes and they've been tested, you merge them back into the main code. This combines your changes with the main project so everyone can use them.

Organization: Using separate branches helps keep the project organized. Once your changes are merged into the main code, you can focus on the next task. By using branches, you and your classmates can work together on the project without interfering with each other's work. It's like having your own workspace to experiment and contribute to the final program!

Must know Git Commands and their meanings

1. Git Configuration

Before you dive into using Git for your projects, it's important to set it up properly with your name and email address. This ensures that when you make contributions to projects, your identity is correctly attributed.

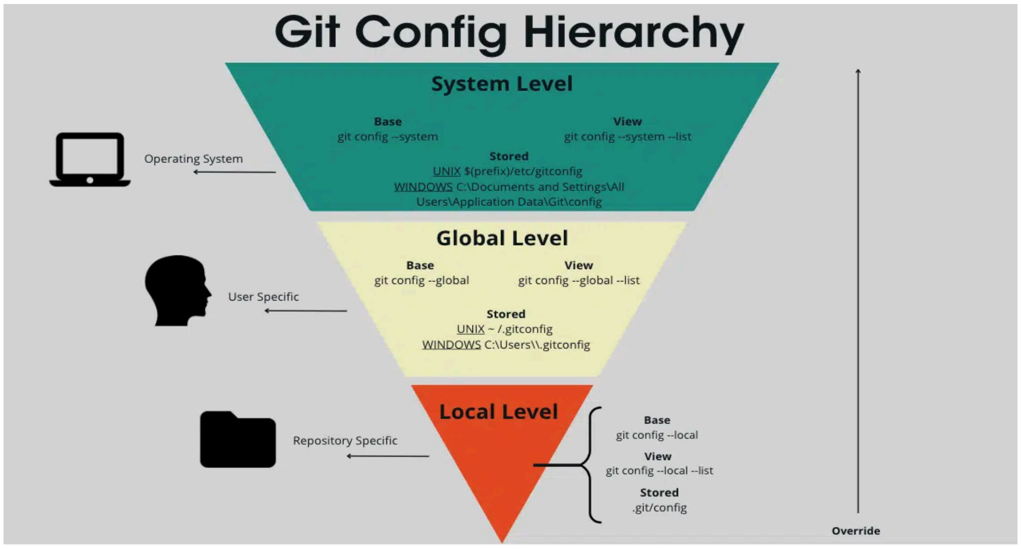
git config --global user.name "Your Name":

This command sets your name for Git. Replace "Your Name" with your actual name. This allows Git to associate your name with the changes you make.

git config --global user.email "you@example.com":

Similarly, this command sets your email address for Git.Your email address is used for communication related to your Git activities, such as notifications and commit logs. git config --global color.ui auto: This command enables colorization for Git output, making it easier to read and understand. It's a helpful visual aid, especially when dealing with complex Git commands or reviewing commit logs.

By configuring these settings globally **(--global),** you ensure that they apply to all Git repositories on your system. Once you've run these commands, Git is set up to recognize you as a contributor to your projects, and your contributions will be properly credited.



2. Starting a Project

1. **Initializing a New Repository**: If you're starting from scratch and want to track changes in your project folder, you'll initialize a new Git repository. This essentially tells Git to start

monitoring the files in that directory for any changes you make. You can do this by navigating to your project folder in the terminal and run :

```
git init
```

After running this command, Git sets up a new repository within the project directory, and you can start adding files, making changes, and committing them to Git's history. **We will understand about how to add and commit files in git later in this document**

2. **Cloning an Existing Repository**: If you're joining an existing project or want to work on a project that's already hosted online (like on GitHub or GitLab), you can clone it to your local machine. Cloning creates a copy of the entire repository, including all its files and commit history, on your computer. You can clone a repository by running:

```
git clone <project_url>
```

This command fetches the entire project and sets up a local copy on your machine, allowing you to start working on it immediately. Whether you're initializing a new repository or cloning an existing one, Git provides you with a solid foundation to manage your project's version control effectively. This ensures that you can track changes, collaborate with others, and maintain a history of your project's development with ease.

3. Day to Day Work

Every day, you'll use Git to manage your coding projects. You'll check what's happening in your project with git status. When you're happy with changes in a file, you git add [file] to tell Git to remember them. Finally, you git commit to save those changes with a short description of what you did and then eventually push your changes to remote, so that others can see those changes as well.This simple workflow keeps your project organized and helps you track your progress.

git status:

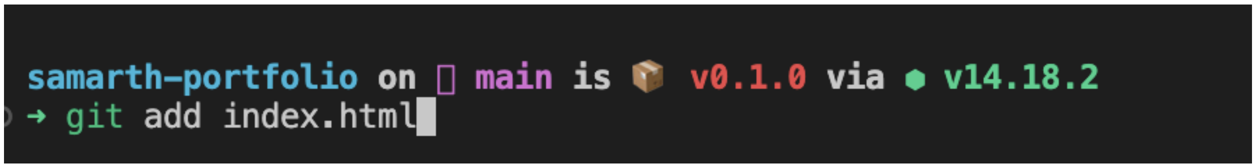
Imagine you're working on a group project, and you want to know what's happening with the files. Git status is like asking the project manager for updates. It tells you which files have been changed, which ones are ready to be saved, and which ones are not being tracked yet.Example:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

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

        style.css

no changes added to commit (use "git add" and/or "git commit -a")
```

git add [file]:

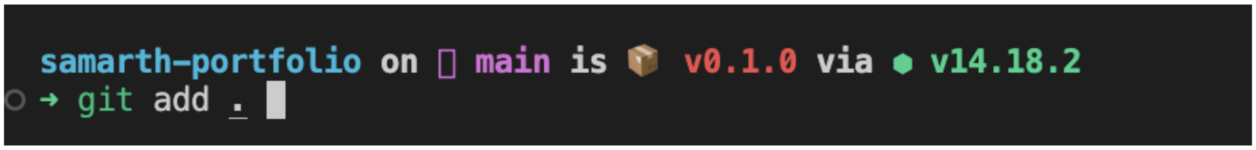
When you're happy with the changes you've made to a file and want to include them in the next save point, you use git add. It's like putting those changes in a box and saying, "Hey, Git, remember these for me!"Example:

If you want to add a specific file



```
samarth-portfolio on ▯ main is 📦 v0.1.0 via ⬢ v14.18.2
➜ git add index.html
```

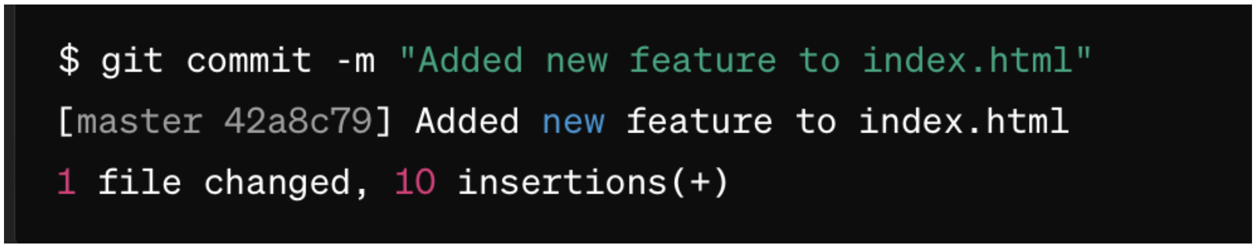
If you want to add all change files in your project directory



```
samarth-portfolio on ▯ main is 📦 v0.1.0 via ⬢ v14.18.2
➜ git add .
```

git commit -m <your message>

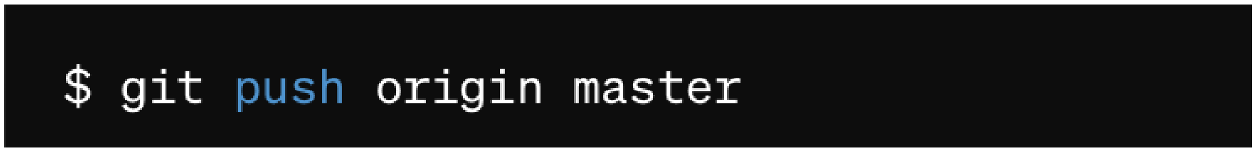
Once you've added all the changes you want to save, you commit them. It's like taking a snapshot of your project at that moment. You also add a short message to describe what you did in this snapshot. Example:



```
$ git commit -m "Added new feature to index.html"
[master 42a8c79] Added new feature to index.html
1 file changed, 10 insertions(+)
```

git push

After you've committed your changes, you might want to share them with others or save them online. Pushing is like sending your saved changes from your computer to a central location, like GitHub or GitLab. It makes your work accessible to teammates and acts as a backup in case something happens to your local files.Example

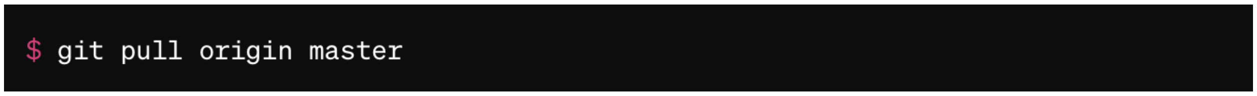


```
$ git push origin master
```

This command sends your committed changes to the "master" branch on the remote repository named "origin." Others can then see and access these changes.

git pull

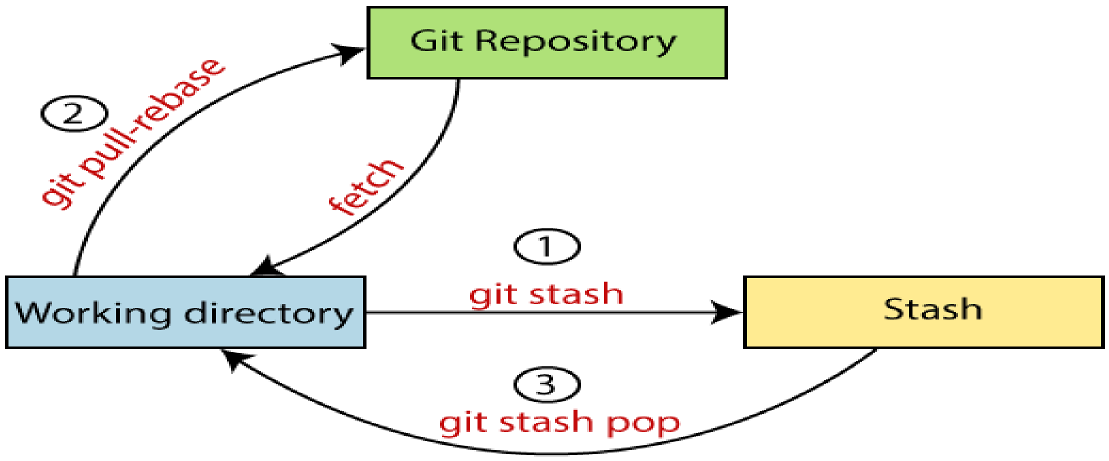
Sometimes, while you're working on a project, others may make changes to the same files you're working on. Git pull helps you stay up-to-date by bringing in those changes from the remote repository to your local copy. It's like fetching the latest updates from the central server and merging them into your own work.Example:



```
$ git pull origin master
```

This command fetches the latest changes from the "master" branch of the remote repository named "origin" and merges them into your local branch. It ensures that your local copy reflects the most current state of the project.

4. Storing Your Work:

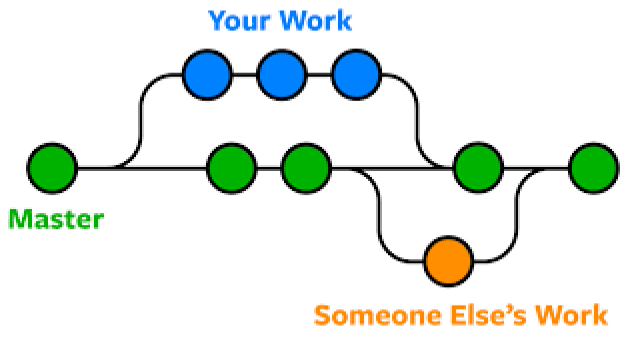


- **git stash:** Imagine you're tidying up your desk by temporarily hiding your work under a mat. Similarly, git stash hides your changes temporarily so you can work on something else.

- **git stash pop:** When you're ready to bring back your hidden work, it's like lifting the mat off your desk. git stash pop retrieves your hidden changes and applies them back to your project

- **git stash drop:** If you've hidden some work that you don't need anymore, you can get rid of it, like throwing away notes you no longer need.

5. Git Branching Model:

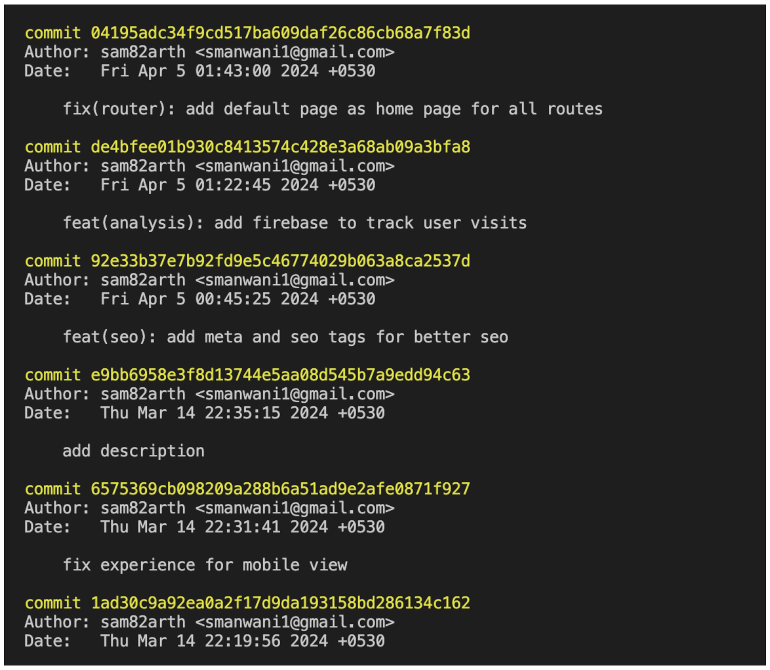


- **git branch [branch_name]:** Creating a branch is like making a copy of your project where you can work on new stuff without messing up the original. git branch [branch_name] creates a new copy (branch) of your project

- **git checkout [branch_name]:** Switching branches is like moving between different versions of your project. git checkout [branch_name] lets you switch to a different version (branch) of your project

- **git merge [branch_name]:** Merging branches is like combining different versions of your project into one. git merge [branch_name] integrates changes from one version (branch) into another

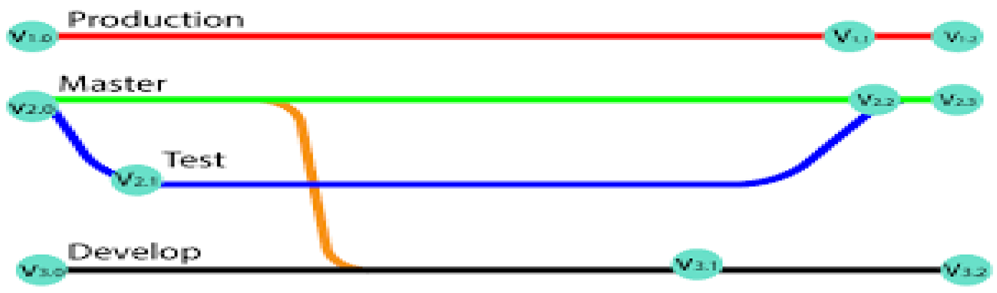
6. Inspect History:



```
commit 04195adc34f9cd517ba609daf26c86cb68a7f83d
Author: sam82arth <smanwani1@gmail.com>
Date:   Fri Apr 5 01:43:00 2024 +0530

    fix(router): add default page as home page for all routes

commit de4bfee01b930c8413574c428e3a68ab09a3bfa8
Author: sam82arth <smanwani1@gmail.com>
Date:   Fri Apr 5 01:22:45 2024 +0530

    feat(analysis): add firebase to track user visits

commit 92e33b37e7b92fd9e5c46774029b063a8ca2537d
Author: sam82arth <smanwani1@gmail.com>
Date:   Fri Apr 5 00:45:25 2024 +0530

    feat(seo): add meta and seo tags for better seo

commit e9bb6958e3f8d13744e5aa08d545b7a9edd94c63
Author: sam82arth <smanwani1@gmail.com>
Date:   Thu Mar 14 22:35:15 2024 +0530

    add description

commit 6575369cb098209a288b6a51ad9e2afe0871f927
Author: sam82arth <smanwani1@gmail.com>
Date:   Thu Mar 14 22:31:41 2024 +0530

    fix experience for mobile view

commit 1ad30c9a92ea0a2f17d9da193158bd286134c162
Author: sam82arth <smanwani1@gmail.com>
Date:   Thu Mar 14 22:19:56 2024 +0530
```

- **git log:** Imagine keeping a diary of all the changes you've made in your project. git log lets you see a list of all your entries (commits) in that diary. Example:
- **git diff** : Comparing versions of your project is like seeing what's changed between two diary entries. git diff shows you the differences between the current state of your project and a previous version. Example:

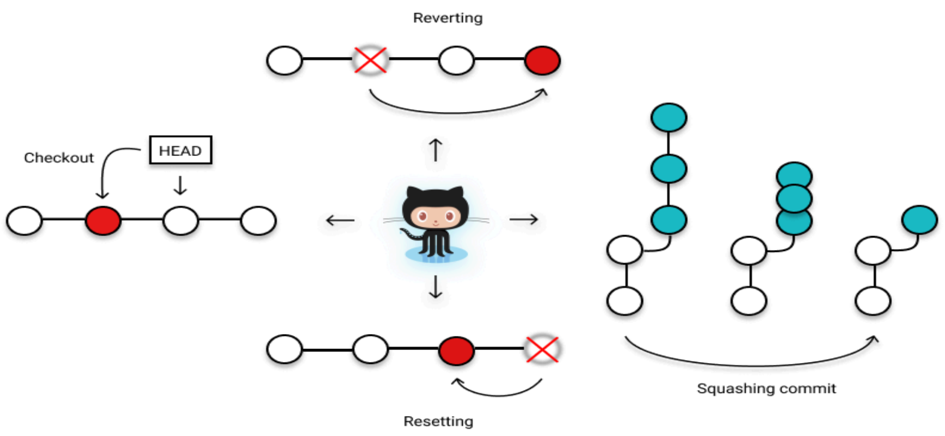
7. Tagging Commits:



- **git tag:** Adding tags to commits is like bookmarking important pages in your diary. git tag lets you see all your bookmarks (tags) in your project.

- **git tag [name] [commit sha]:** Creating a tag is like putting a special bookmark on a particular page in your diary. git tag [name] [commit sha] creates a new bookmark (tag) at a specific point in your project's history.

8. Reverting Changes:



9.

- **git reset:** If you've made a mess in your project, you can start over by resetting it to a previous state, like rewinding to an earlier point in your diary.
- **git revert [commit sha]:** Making a new change that undoes an old one is like writing a correction in your diary. git revert [commit sha] creates a new entry (commit) that reverses the changes made in a previous one.

10. Synchronizing Repositories:

- **git fetch [remote]:** Fetching updates from collaborators' repositories is like getting mail from your friends. git fetch [remote] gets the latest updates from a remote repository but doesn't merge them into your local version.

GIT CHEAT SHEET

1. **git init**: Initializes a new Git repository in the current directory.
2. **git clone [repository_url]**: Copies an existing Git repository from a specified URL to your local machine.
3. **git add [file]**: Adds a file to the staging area for the next commit.
4. **git commit -m "[message]"**: Records changes to the repository along with a descriptive message.
5. **git status**: Displays the state of the working directory and the staging area.
6. **git diff**: Shows the differences between the working directory and the staging area.
7. **git diff --staged**: Shows the differences between the staging area and the last commit.
8. **git log**: Displays the commit history.
9. **git branch**: Lists all local branches in the current repository.
10. **git branch [branch_name]**: Creates a new branch.
11. **git branch -d [branch_name]**: Deletes a specified branch.
12. **git checkout [branch_name]**: Switches to the specified branch.
13. **git checkout -b [branch_name]**: Creates a new branch and switches to it.
14. **git merge [branch_name]**: Merges changes from the specified branch into the current branch.
15. **git remote**: Lists remote repositories associated with the current repository.
16. **git remote add [name] [url]**: Adds a new remote repository.
17. **git push [remote] [branch]**: Uploads local branch commits to the remote repository.
18. **git push -u [remote] [branch]**: Sets the upstream branch for the current local branch.
19. **git pull [remote] [branch]**: Downloads changes from the remote repository and incorporates them into the current branch.

20. **git fetch [remote]**: Downloads changes from the remote repository without merging them into the current branch.
21. **git reset [file]**: Unstages the specified file, but leaves its contents unchanged.
22. **git reset --hard**: Resets the staging area and working directory to match the most recent commit.
23. **git rm [file]**: Deletes the specified file from the working directory and stages the deletion.
24. **git mv [old_file] [new_file]**: Renames a file and stages the change.
25. **git stash**: Temporarily shelves changes and removes them from the working directory.
26. **git stash pop**: Applies the most recently stashed changes to the working directory.
27. **git stash list**: Lists all stashed changes.
28. **git stash drop**: Deletes the most recently stashed changes.
29. **git cherry-pick [commit]**: Applies the changes introduced by the specified commit to the current branch.
30. **git rebase [base]**: Reapplies commits on top of another base tip.
31. **git rebase -i [base]**: Rebases interactively; lets you squash, edit, and reorder commits.
32. **git tag**: Lists tags in the repository.
33. **git tag [tag_name]**: Creates a new tag at the current commit.
34. **git tag -d [tag_name]**: Deletes the specified tag.
35. **git show [tag_name]**: Displays the tag along with the commit details it points to.
36. **git fetch --tags**: Downloads all tags from the remote repository.
37. **git push [remote] [tag_name]**: Publishes the specified tag to the remote repository.
38. **git push [remote] --tags**: Publishes all local tags to the remote repository.
39. **git describe [commit]**: Outputs the most recent tag reachable from a commit.
40. **git config**: Lists or sets configuration options.

41. **git config --global user.name "[name]"**: Sets the name to be used with your commits.
42. **git config --global user.email "[email]"**: Sets the email to be used with your commits.
43. **git config --global alias.[alias_name] "[git_command]"**: Sets a custom alias for a Git command.
44. **git log --oneline**: Displays the commit history in a condensed format.
45. **git log --graph**: Displays the commit history as a graph.
46. **git log --author="[author]"**: Filters the commit history by author.
47. **git log --grep="[pattern]"**: Filters the commit history by commit message.
48. **git log -p**: Shows the patch representing each commit.
49. **git log --stat**: Shows the files that were modified in each commit, along with the number of lines that were added or removed.
50. **git reflog**: Displays a record of all commits that are or were referenced in the repository.
51. **git blame [file]**: Shows who last modified each line of a file and when.
52. **git clean -n**: Shows a list of untracked files that would be removed by git clean -f.
53. **git clean -f**: Removes untracked files from the working directory.
54. **git bisect start**: Starts the binary search for the commit that introduced a bug.
55. **git bisect bad**: Marks the current commit as bad.
56. **git bisect good [commit]**: Marks the specified commit as good.
57. **git bisect reset**: Resets the binary search and returns to the original branch.
58. **git fsck**: Verifies the connectivity and validity of the objects in the database.
59. **git grep "[pattern]"**: Searches the working directory for lines matching the specified pattern.
60. **git ls-files**: Lists all files in the index.
61. **git merge --abort**: Aborts the current merge and resets the working directory to the state before the merge.
62. **git merge --continue**: Continues the current merge after resolving any conflicts.

63. **git pull --rebase [remote] [branch]**: Downloads changes from the remote repository and rebases the current branch on top of the fetched changes.
64. **git submodule init**: Initializes the submodule configuration.
65. **git submodule update**: Updates the registered submodules to match the commit specified in the index of the containing repository.
66. **git clean -xfd**: Removes all untracked files, including directories, from the working directory.
67. **git log --author="[author]" --after="[date]" --before="[date]"**: Filters the commit history by author and date range.
68. **git log --graph --decorate --oneline**: Displays a compact and decorated commit history graph.
69. **git log --pretty=format:"%h - %an, %ar : %s"**: Displays a customized log format with abbreviated commit hash, author name, relative commit date, and subject.
70. **git log --since="[date]" --until="[date]"**: Filters the commit history by commit dates.
71. **git config --global core.editor "[editor]"**: Sets the default text editor for Git messages.
72. **git reflog expire --expire=[time] [branch]**: Removes entries older than the specified time from the reflog of the specified branch.
73. **git reset --soft [commit]**: Moves the current branch tip to the specified commit and stages the changes.
74. **git reset --mixed [commit]**: Moves the current branch tip to the specified commit and unstages the changes.
75. **git reset HEAD [file]**: Unstages the specified file without changing the working directory or the last commit.
76. **git tag -a [tag_name] -m "[message]" [commit]**: Creates an annotated tag at the specified commit with a message.
77. **git tag -l "[pattern]"**: Lists tags that match the specified pattern.
78. **git show-branch**: Displays the commit history with branches.

79. **git shortlog**: Summarizes git log output.
80. **git shortlog -s**: Shows just the commit count and the author name.
81. **git shortlog -n**: Sorts the summary by the number of commits per author in descending order.
82. **git archive --format=[format] --output=[file] [branch]**: Creates an archive of the specified branch.
83. **git cherry-pick --continue**: Continues the cherry-pick operation after resolving conflicts.
84. **git cherry-pick --quit**: Cancels the cherry-pick operation.
85. **git cherry-pick --abort**: Aborts the cherry-pick operation and restores the original state.
86. **git commit --amend**: Adds staged changes to the previous commit.
87. **git commit --amend -m "[message]"**: Amends the previous commit with a new message.
88. **git commit --amend --no-edit**: Adds staged changes to the previous commit without changing the commit message.
89. **git commit --fixup [commit]**: Creates a fixup commit to be combined with the specified commit during a rebase.
90. **git commit --squash [commit]**: Creates a squash commit to be combined with the specified commit during a rebase.
91. **git rebase -i --autosquash [base]**: Rebases interactively and automatically applies fixup and squash commits marked as autosquash.
92. **git remote show [remote]**: Displays information about a remote repository.
93. **git remote update**: Fetches updates from all remote repositories.
94. **git fetch [remote] [branch] --prune**: Fetches updates from the specified remote repository and removes any remote-tracking branches that no longer exist on the remote.
95. **git stash apply [stash]**: Applies the specified stash to the working directory.
96. **git stash apply --index [stash]**: Applies the specified stash to the working directory and the staging area.

97. **git stash show [stash]**: Displays the changes contained in the specified stash.
98. **git stash show -p [stash]**: Displays the patch representing the changes contained in the specified stash.
99. **git stash drop [stash]**: Deletes the specified stash.
100. **git stash clear**: Deletes all stashed changes.

BONUS - YOUTUBE VIDEOS

How Git Works: Explained in 4 Minutes - overview
Git Tutorial for Beginners: Learn Git in 1 Hour - in depth