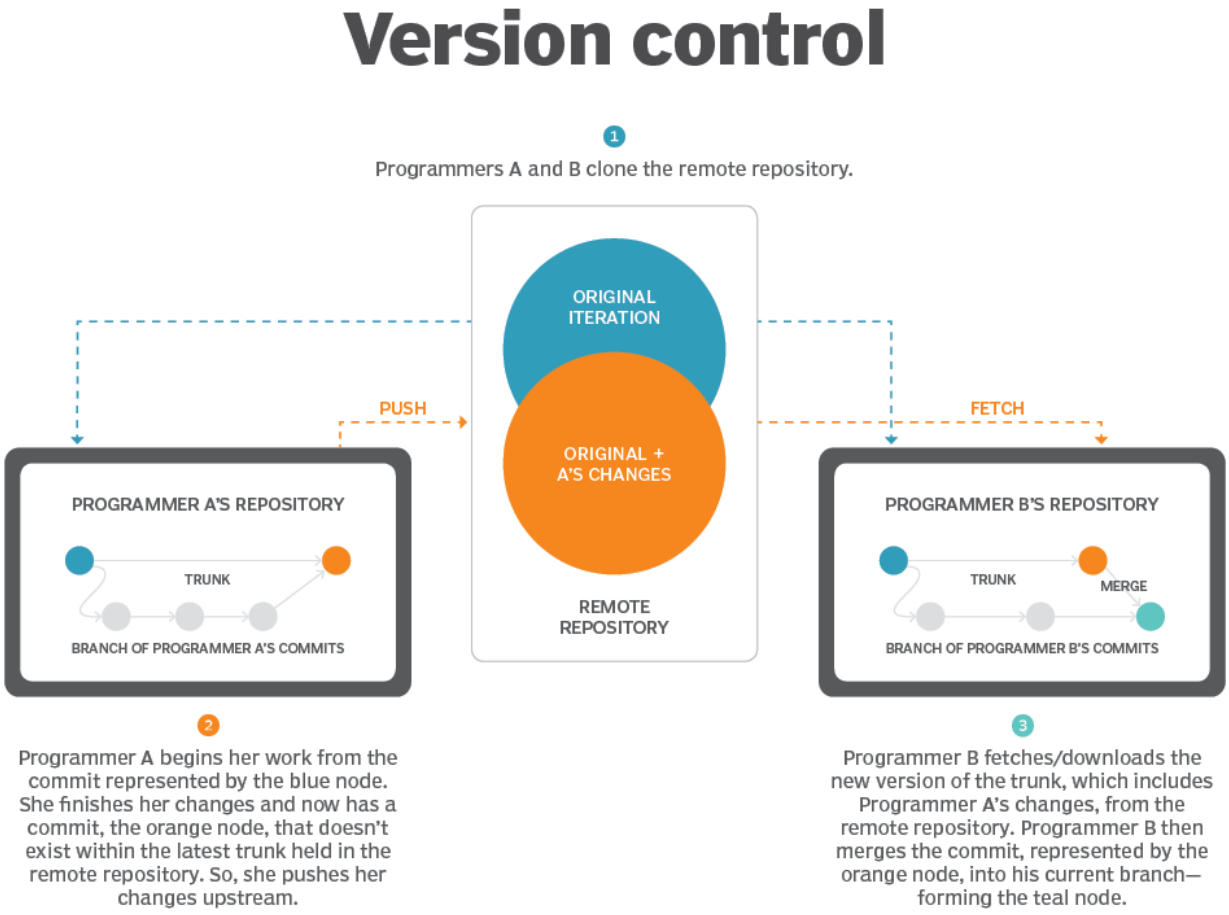
Git & GitHub

Introduction

What is Version Control System (VCS)?

- A VCS tracks and records changes to any file (or a group of files) allowing you to recall specific iterations later or as needed.
- VCSs are sometimes called **source code management (SCM)** or **revision control systems (RCS)**.
- Version control allows numerous team members to work collaboratively on a project, even if they're not in the same room or even country.

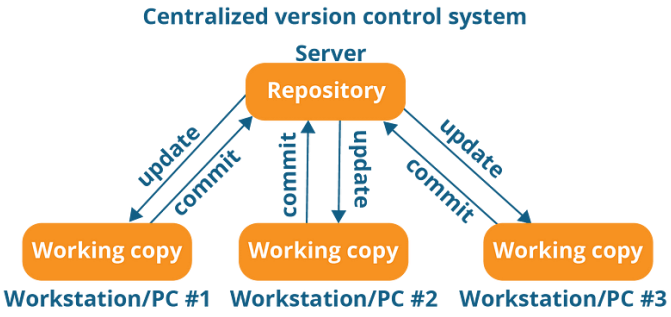


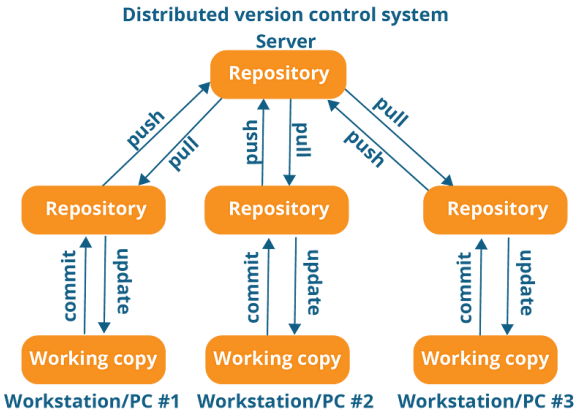
What is Git?

- Git is a distributed version control system that tracks changes in any set of computer files, usually used for coordinating work among programmers who are collaboratively developing source code during software development.
- Its goals include speed, data integrity, and support for **distributed**, **non-linear workflows** (thousands of parallel branches running on different computers).
- There are many popular offerings of Git repository services, including GitHub, SourceForge, Bitbucket and GitLab.

Centralized and Distributed Version Control Systems

- Both centralized and distributed systems, such as Git, perform the same function.
- The key difference between the two is that centralized systems have a central server where team members push the latest versions of their work.
- You can think of it somewhat like having a single central project that everyone shares.
- With distributed VCSs, team members have **a local copy (clone)** of the entire project's history on their own device, so they don't need to be online to make changes or work on their code.
- Instead of a centralized server, they source this clone from an **online repository**.
- When developers work with Git, every team member's clone of the project is a repository that can contain all changes from the beginning of the project.





Following are the key differences between both:

1. **Architecture:**
   - **Git**:
     - Git is a distributed version control system, meaning that every developer has a complete copy of the entire repository, including the entire history of changes.
     - This allows developers to work offline and perform most version control operations locally without needing to connect to a central server.
   - **CVS**:
     - CVS is a centralized version control system, where there is a single central repository that stores the entire history of changes.
     - Developers check out files from the central repository to make changes and commit them back to the repository when done.
2. **Branching and Merging:**
   - **Git**:
     - Git provides powerful branching and merging capabilities, allowing developers to create lightweight branches for feature development or experimentation.
     - Merging changes between branches in Git is generally fast and efficient.
   - **CVS**:
     - CVS supports branching and merging but with limited capabilities compared to Git.
     - Branching in CVS is heavyweight and can be complex, and merging changes between branches can sometimes be difficult and error prone.
3. **Performance:**
   - **Git**:
     - Git is known for its performance and speed, especially for operations like branching, merging, and committing changes.
     - Since most operations are performed locally, developers can work efficiently even with large repositories and complex history.

- **CVS**:
  - CVS can suffer from performance issues, especially with large repositories and complex branching and merging operations.
  - Since most operations involve interaction with a central server, performance can be affected by network latency and server load.

4. **Workflow:**
   - **Git**:
     - Git encourages flexible and decentralized workflows, where developers can work independently on their local branches and merge changes into the main branch (e.g., `main`) when ready.
     - This allows for greater collaboration and parallel development.
   - **CVS**:
     - CVS follows a more centralized and linear workflow, where developers check out files from the central repository, make changes, and commit them back to the repository.
     - Collaboration is more centralized, and developers need to coordinate changes more closely.
5. **Features:**
   - **Git**:
     - Git offers a rich set of features, including branching and merging, distributed development, local history, stashing, rebasing, and more.
     - It also supports a wide range of workflows and integrations with other tools and services.
   - **CVS**:
     - CVS provides basic version control features, such as branching, merging, tagging, and file locking.
     - However, it lacks some of the more advanced features and capabilities offered by modern version control systems like Git.

Why is a Version Control System like Git needed?

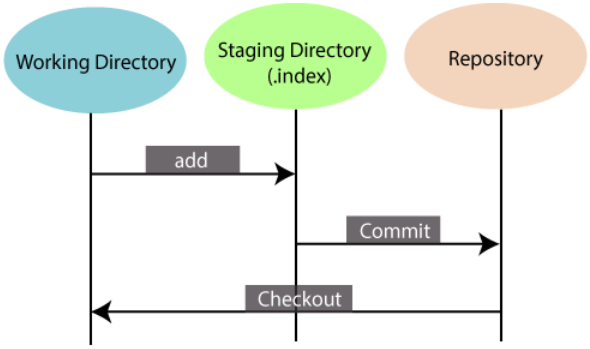
- Real life projects generally have multiple developers working in parallel. So, a version control system like Git is needed to ensure there are no code conflicts between the developers.
- Additionally, the requirements in such projects change often. So, a version control system allows developers to revert and go back to an older version of the code.
- Finally, sometimes several projects which are being run in parallel involve the same codebase. In such a case, the concept of branching in Git is very important.

Features of Git

- **Open Source**
  - Git is an open-source tool. It is released under the GPL (General Public License) license.
- **Scalable**
  - Git is scalable, which means when the number of users increases, Git can easily handle such situations.
- **Distributed**
  - One of Git's great features is that it is distributed. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository.
  - Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project.
  - We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.
- **Security**
  - Git is secure. It uses the SHA1 (Secure Hash Function) to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout.

- It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

- **Speed**
  - Git is very fast, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a huge speed. Also, a centralized version control system continually communicates with a server somewhere.
  - Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs. The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages.
  - Git was developed to work on the Linux kernel; therefore, it is capable enough to handle large repositories effectively. From the beginning, speed and performance have been Git's primary goals.
- **Supports non-linear development**
  - Git supports seamless branching and merging, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.
- **Branching and Merging**
  - Branching and merging are the great features of Git, which makes it different from the other SCM tools (Source code Management). Git allows the creation of multiple branches without affecting each other. We can perform tasks like creation, deletion, and merging on branches, and these tasks take only a few seconds. Below are some features that can be achieved by branching:
    - We can create a separate branch for a new module of the project, commit and delete it whenever we want.
    - We can have a production branch, which always has what goes into production and can be merged for testing in the test branch.
    - We can create a demo branch for the experiment and check if it is working. We can also remove it if needed.
    - The core benefit of branching is if we want to push something to a remote repository, we do not have to push all our branches. We can select a few of our branches, or all of them together.
- **Data Assurance**
  - The Git data model ensures the cryptographic integrity of every unit of our project. It provides a unique commit ID to every commit through a SHA algorithm. We can retrieve and update the commit operation by commit ID.
- **Staging Area**
  - The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion.
  - When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the change.
  - Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses **a file called index**.

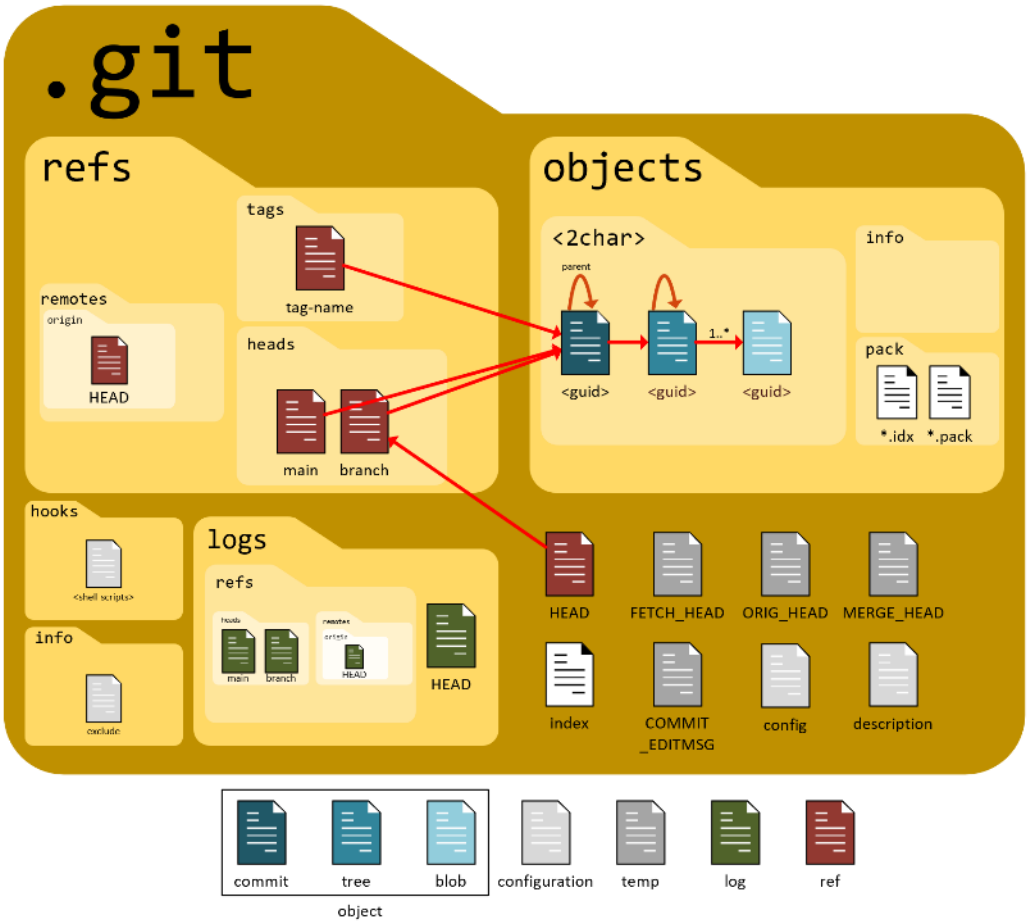


    - Another feature of Git that makes it apart from other SCM tools is that it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.
- **Maintain the clean history**
    - Git facilitates with `git rebase`; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.
- **Compatibility with existing systems and protocols**
    - Repositories can be published via Hypertext Transfer Protocol Secure (HTTPS), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), or a Git protocol over either a plain socket or Secure Shell (SSH).
    - Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Subversion repositories can be used directly with git-svn.

Git Concepts and Architecture

Repository (Repo)

- A repository is a directory where all the files for a particular project are stored.
- A Git repository is the `.git/` folder inside a project.
- This repository tracks all changes made to files in your project, building a history over time.
- Meaning, if you delete the `.git/`, you delete your project's history.
- When you initiate Git in a directory (`git init`), it becomes a local repository.



- انا عاوز الgit يكون OS independent عشان كدة هخلى الstructure بتاعه عبارة عن فولدر عادى اقدر احطه في اى OS ويفهمه والفايلات بتاعته نفسها كمان معظمها هتكون حاجات بسيطة كمان للOS مش هخلى فيه يعنى engine او database او كدة ، مش بس كدة انا كمان خليته portable عن طريق انى خليت فولدر الgit نفسه (`.git/`) موجود جوة الفولدر بتاع الشغل عشان مهما انقل فولدر الشغل دة يكون حاجة الgit كلها جواه.

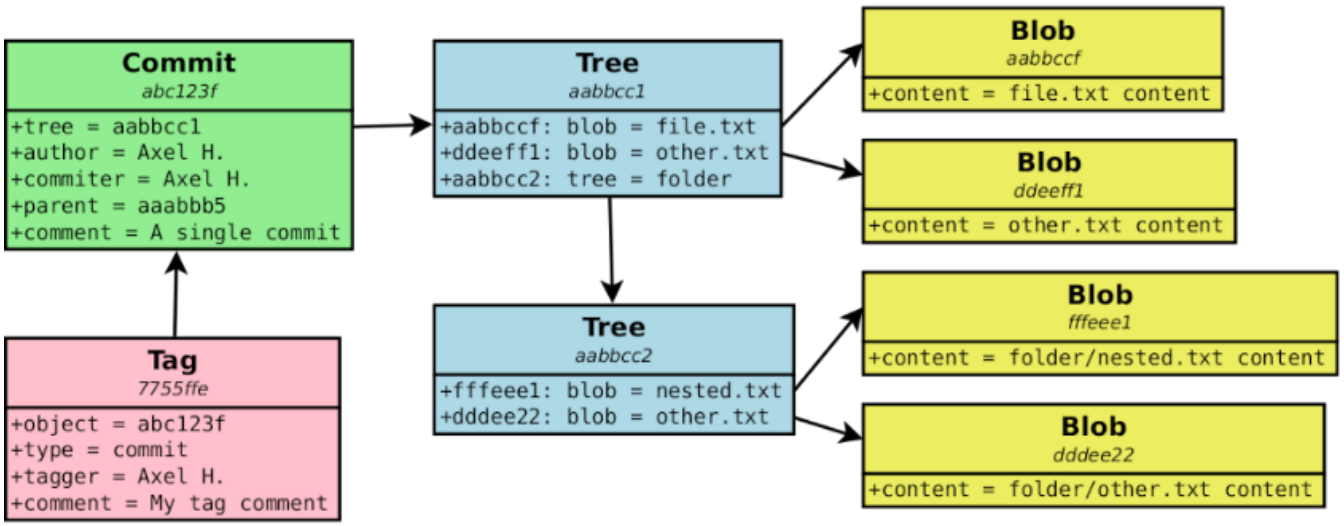
`git status`	Show Status. `git status --short  || git status -s` **Note:** Short status flags are: • `??` - Untracked files • `A` - Files added to stage • `M` - Modified files • `D` - Deleted files
`git help`	`git command -help` ----> See all the available options for the specific command. `git help --all` ----> See all possible commands.
`git init`	Create git repository in current directory. `git init <RepoName>` ----> Create a new repo and initialize .git/ inside it.
`git config`	`git config --list` ----> List all available settings. **Note:** Use `--global` to set the username and e-mail for every repository on your computer. If you want to set the username/e-mail for just the current repo, you can remove global.

```
git config --global user.name                  ----> Return the existing value.
git config --global user.name "Mohaned"        ----> Setting new value.
git config --global --unset user.name ""       ----> Only clearing the config option value.
git config --global --unset user.name          ----> Removing the config option from the list.
git config --global --edit                     ----> Editing config file through an editor.
git config --global alias.cm "commit -m"       ----> Aliasing "commit -m" with cm.
git config --global --unset alias.NAME
```

Objects

- In Git, every commit, every tree, and every file are saved in the objects folder as a hash value.
- Generally, Git objects consist of **4 types**:
  - **Blob (file):** Blob stores the contents of the file. Whenever we commit our files the first type of objects that are created are the blob objects.
  - **Tree (directory):** Tree objects contain a list of all files in our repository with a pointer to the blob object assigned to them.
  - **Commit:** Git creates a commit object that has a pointer to its tree object. The commit object contains:
    - Tree object hash
    - Parent commit hash
    - Author
    - Committer
    - Date
    - Message
  - **Tag:** A tag object contains an object name, object type, tag name, the name of the person who created the tag, and a message.
- There is a unique hash value for every object which helps Git to where it is located.
- Git generates a 40-character checksum (SHA-1) hash for every object and the first two characters of that checksum are used as the directory name and the other 38 as a file name.
- انا عاوز اعمل Unique ID لكل object عندى عشان اعرف اعمل tracking ليه ، الgit بيستخدم مفهوم الhashing عشان يميز بين كل object والتانى ، ودة عن طريق انه بيدخل لHash function أربع حاجات:

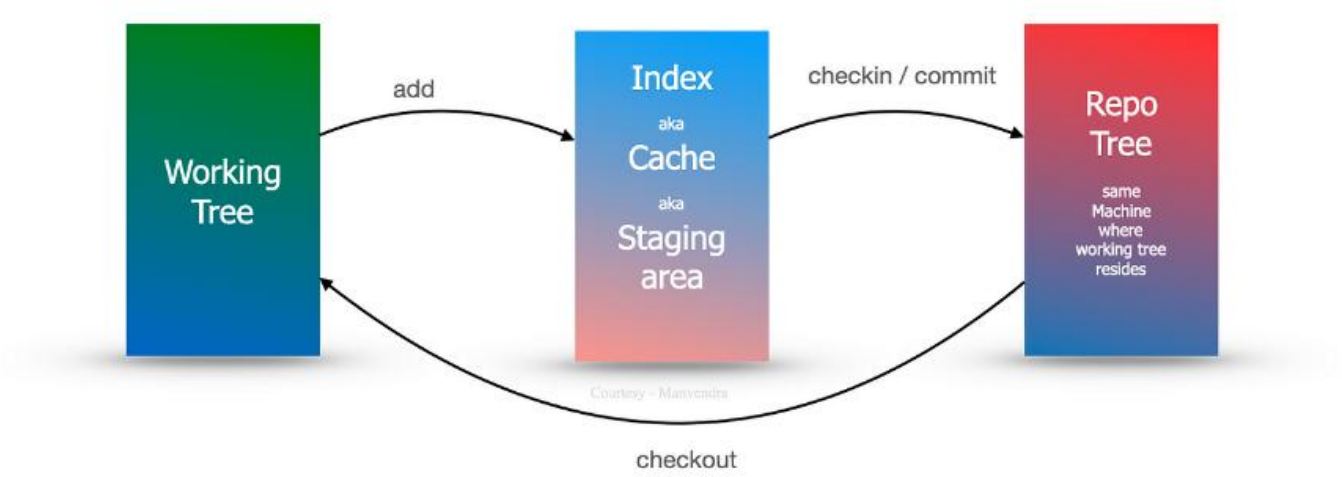
Object Type – Object Size – Null Character – Object Content



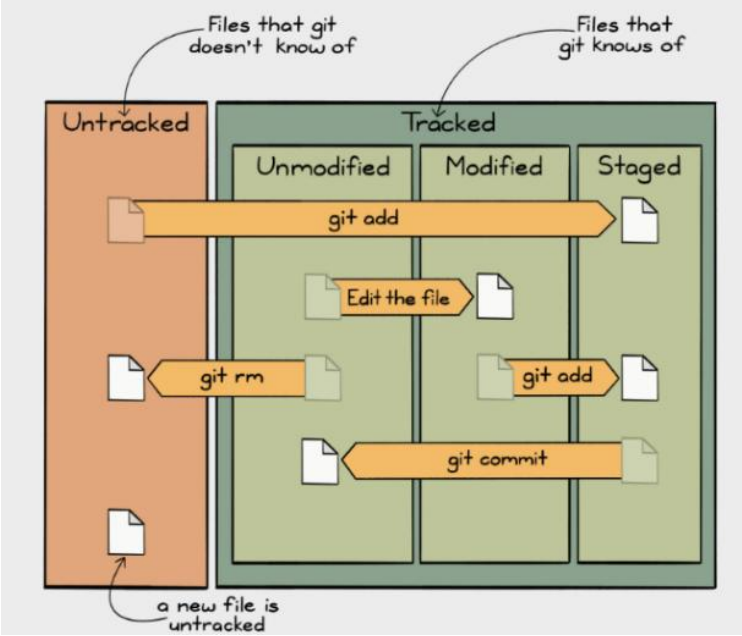
Staging area

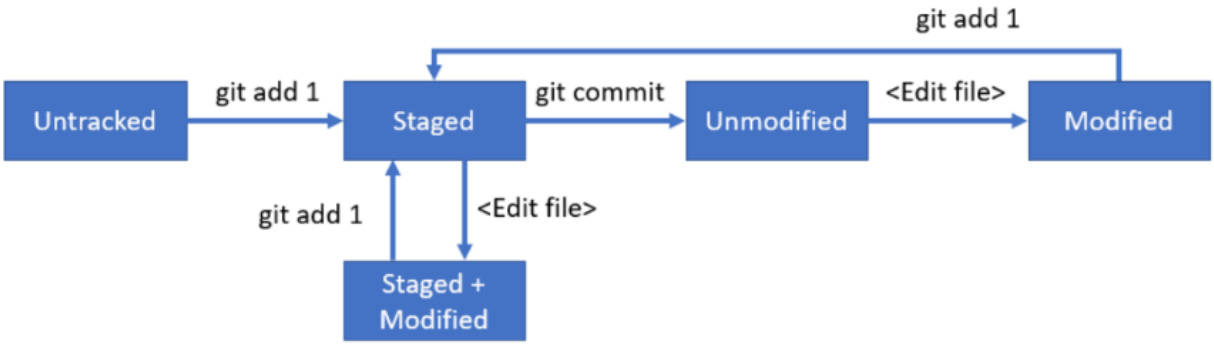
- Before finalizing changes with a commit, you first "stage" them.
- The staging area is like a draft space where you prepare your changes before committing them.

- الgit شغال بنظام الـ3-tree architecture ، ومرحلة الstaging دى ممكن استفاد منها انى مثلا بفضل اعدل الملفات كل واحد لوحده وكل ما اخلص واحد اعمله staging وفى الاخر خالص اعمل commit مرة واحدة عشان دة يبقى version واحد بس جديد.
- مرحلة الstaging دى هي ملف اسمه index بيتكتب فيه الهاش بتاع الفايل لما بحوله من untracked لtracked عن طريق `git add`



- الfile status في الgit بيكون حاجتين ياما untracked يعنى لسه جديد ومعملتش لسه `git add` ، ياما tracked وبيكون حالة من اتنين modified او unmodified.





`git add`	Add files to staging area.
	
	`git add <File> <File>`
	`git add *.extension`
	`git add *`

`git ls-files`	It shows files that are tracked by Git, i.e., files in the index (staging area) or the working directory that are not ignored. • `--cached` Show files in the index (staged for commit). • `--modified` Show files that are modified in the working directory but not staged. • `--deleted` Show files that are deleted from the working directory but still tracked. • `--others` Show files in the working directory that are not tracked and not ignored. • `--ignored` Show files that are ignored by `.gitignore`. • `--stage` Show files along with their staging information (e.g., file mode, object ID, and stage number).
`git cat-file`	Provide contents or details of repository objects. `git cat-file <type> <object-hash>` ----> Show object content. `git cat-file -t eff6b` ----> Show object type. `git cat-file -s eff6b` ----> Show object size. `git cat-file -p eff6b` ----> Show object content.
`git clean`	Removes untracked files and directories from the working directory. It is helpful for cleaning up a repository by deleting files that are not tracked by Git or ignored by `.gitignore`. By default, Git is globally configured to require that `git clean` be passed a "force" option to initiate. `git clean -n` ----> Show what untracked files would be deleted. `git clean -f` ----> Force deletes untracked files. `git clean -df` ----> Force deletes untracked files and directories. `git clean -di` ----> Open interactive clean mode that will act on directories also.

`git restore`	Discards changes in the working directory or staging area.
	Can unstage files that have been added to the index (staging area).
	Restores a file or directory to a previous commit.
	
	`git restore [options] <path>`
	
	• `--source=<tree>` Specify the source (commit or branch) from which to restore the file.
	• `--staged` Remove changes from the staging area (unstage files).
	• `--worktree` Discard changes in the working directory (default).
	• `--patch` Interactively restore parts of a file.
	• `-s` or `--source` Specify the commit to restore from. Defaults to HEAD.
	
	`git restore --staged css\first.css` ----> Unstage specified file.
	`git restore --staged *` ----> Unstage all files.
	`git restore README.md` ----> Resets README.md to the version in the latest commit (HEAD).
	`git restore --staged --worktree <file>` ----> Discard changes in both the working directory and staging area for a specific file.
	`git restore --source=abc123 README.md` ----> Restores README.md to its state in the commit with hash `abc123`

Commits

- Every change or set of changes that you finalize in Git is called a **commit**.
- Each commit has a unique ID (a SHA-1 hash) that allows Git to keep track of the changes and the order in which they were made.
- 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.
- لما باجى اعمل عملية الcommit بيتعمل 3 أنواع ملفات جوة الgit repository :
  - الأول هو commit object ودة بيكون فيه معلومات عن عملية الcommit اللى عملتها

```
$ git log
commit d56e798637ca38778e394d9b8262296b082de836 (HEAD -> master)
Author: Mohaned <manoahmad97@gmail.com>
Date:   Thu Apr 18 12:48:47 2024 +0200
```

```
$ git cat-file -p d56e
tree e5e8fad15bc1c346c84bec724ba754960e1a3a65
author Mohaned <manoahmad97@gmail.com> 1713437327 +0200
committer Mohaned <manoahmad97@gmail.com> 1713437327 +0200

My first commit
```

  - التانى هو tree object بيكون فيه وصف الhierarchy بتاعة الworking directory لما جيت اعمل commit

```
$ git cat-file -p e5e8
100644 blob e7c9751bb0960b9394314288a672c7edc381c074    file1.txt
```

  - والتالت هو blob objects اللى اتعمل ليها تعديل.

```
$ git cat-file -p e7c9
Hi, Git!
```

- المفروض بعد ما بعمل commit كل الفايلات بتكون unmodified لأن خلاص اتعمل ليهم كلهم snapshot في الgit repository ، لو جيت بقى بعدها عدلت في ملف وعملت git status هلاقيه اصبح Modified باللون الأحمر معناه ان الملف اصبح مختلف عن الtracking بتاعه في الstaging ، بعد ما بعمل git add بيبقى Modified باللون الأخضر معناه ان الملف اصبح مختلف عن اللى موجود في الgit repository. اللى هو مختلف عن حالته في اخر commit يعنى

```
$ git status -s
 M file1.txt
```

```
$ git status -s
M  file1.txt
```

- يبقى كدة لما اجى ابدء ادور أول حاجة اروح للcommit عشان هو اللى wrapper بتاع التعديل كله ، بعد كدة اروح للtree بعد كدة للblobs
- كل commit المفروض يكون فيه attribute بيشاور على الparent بتاعه لأنهم مرتبطين ببعض عن طريق linked list ، ما عدا اول commit بيكون الparent بتاعها null وبنسميها root commit.
- مجموعة الcommits دى كلها على بعضها تعملى الbranch والdefault بيكون الmaster او الmain بقى حديثا.

`git commit`	Commit changes in staging area.
	`git commit -m “A message to be shown for the commission”`
	
	Commit changes directly without staging area (**Should be tracked first**):
	`git commit -a -m “A message to be shown for the commission”`

`git log`	Show commit logs. `git log --oneline` ----> Condenses each commit to a single line. `git log --decorate` ----> Displays the references (branches, tags, etc) that point to each commit. `git log --graph` ----> Draws an ASCII graph representing the branch structure of the commit history. `git log -3` ----> Displays only the 3 most recent commits. `git log --after="2014-7-1" --before="2014-7-4"` ----> Displays commits in specific period. `git log --author="John"` ----> This displays all commits whose author includes the name John.

`git shortlog`	It groups each commit by author and displays the first line of each commit message.
`git reflog`	View the history of changes made to the references in a Git repository, such as HEAD, branches, or tags. It tracks all movements of HEAD (the current commit pointer), including commits, resets, checkouts, merges, and rebases, even if those actions aren't reflected in the commit history. Useful for recovering commits that are no longer part of a branch (e.g., after a `reset --hard`).

Feature	git reflog	git log
Scope	Tracks changes to HEAD and references.	Displays commits in the commit history.
Includes Orphaned Commits	Yes	No
Use Case	Debugging, recovery, and undoing operations.	Viewing commit history and authorship.

`git reset`	Undo changes by resetting the state of the current branch and optionally the staging area and/or working directory. It can be used to unstage files, undo commits or reset the branch's history to a previous state. `git reset [<mode>] [<commit>]` **Modes of Operation** • `--soft` Moves the branch pointer (HEAD) to the specified commit but keeps changes in the staging area and working directory. • `--mixed` (default) Moves the branch pointer to the specified commit and unstages files while keeping changes in the working directory. • `--hard` Moves the branch pointer to the specified commit and removes changes from the staging area and working directory. • `--merge` Similar to `--hard` but keeps uncommitted changes safe if they do not overlap with the reset commit. • `--keep` Resets the branch pointer but keeps uncommitted changes that are not conflicting. `git reset <file>` ----> Unstage files that were added to the staging area `git reset --soft HEAD~1` ----> Undo the last commit but keep changes in the staging area `git reset --mixed HEAD~1` ----> Undo the last commit and unstage the changes `git reset --hard HEAD~1` ----> Undo the last commit and remove changes from both the staging area and working directory `git reset --hard <commit-hash>` ----> Reset the branch to a specific commit

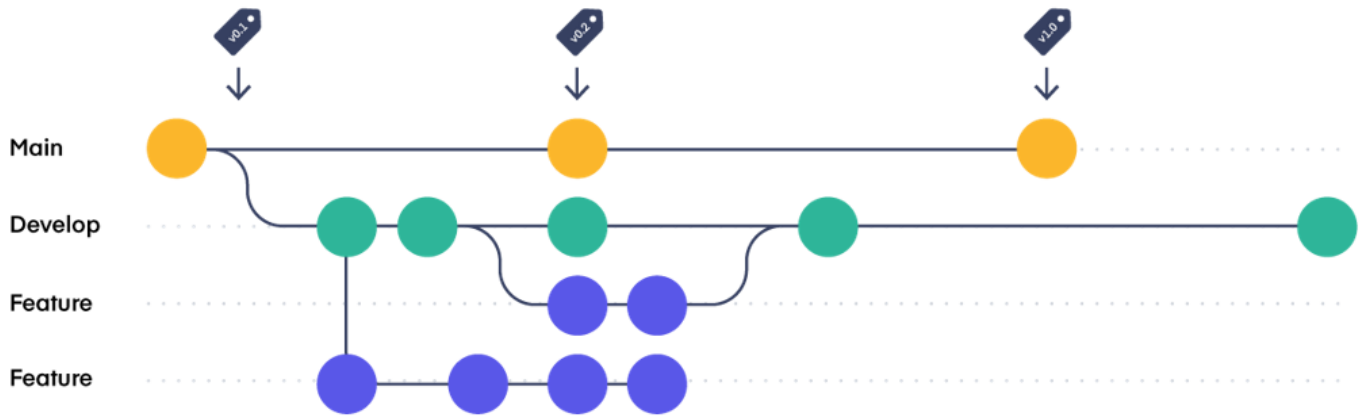
`git revert`	Create a new commit that undoes the changes of a specified commit.
	
	Unlike `git reset`, which removes commits from the history, `git revert` preserves the history by adding a new commit that negates the changes.
	
	`git revert <commit-hash>`
	`git revert <commit1> <commit2>`
	
	If you made a mistake while reverting, you can revert the revert:
	`git revert <revert-commit-hash>`
	
	If there are conflicts during the revert process, resolve them and then complete the revert with:
	`git revert --continue`

Branches

- Branching is one of the most powerful features of Git, enabling developers to create independent lines of development within a repository.
- Branches allow you to work on new features, fix bugs, or experiment with changes without affecting the main codebase.
- Git's lightweight and efficient branching model makes it a core tool in version control workflows like Git Flow and trunk-based development.
- A branch in Git is simply **a pointer to a specific commit**.
- In this sense, a branch represents the tip of a series of commits—it's **not a container** for commits.
- By default, every repository starts with a branch named `main` (or `master` in older conventions).
- New branches are created to diverge from the main development line and can later be merged back.

Benefits of Branching

- **Isolated Development**: Work on a feature or bug fix independently.
- **Parallel Workflows**: Multiple developers can work on different branches simultaneously.
- **Experimentation**: Try out new ideas without risking the stability of the main branch.
- **Collaboration**: Use branches for pull requests, code reviews, and teamwork.



Creating remote branches

- To create a remote branch from your local branch in Git:
  - Check the branch you are on (ensure you are on the correct local branch):

```
git branch
```

  - Create and push the local branch to the remote repository:

```
git push origin <local-branch-name>:<remote-branch-name>
```

    - If you want the remote branch to have the same name as the local branch:

```
git push origin <local-branch-name>
```

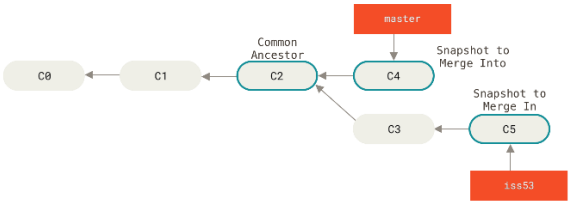
  - Set the upstream tracking branch (optional but recommended):

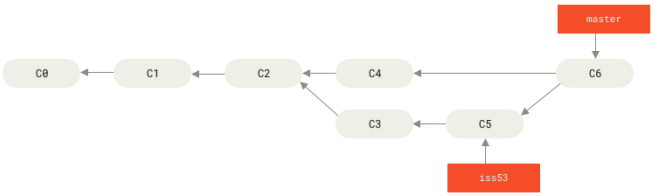
```
git push --set-upstream origin <remote-branch-name>
```

    - This command sets the upstream tracking for your local branch, making it easier to pull and push changes in the future.

Merging

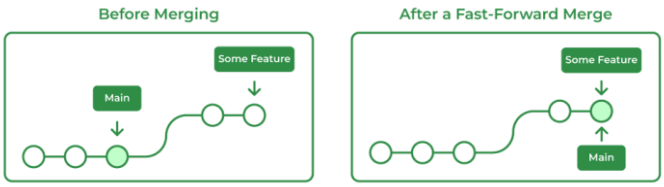
- Once you're done with your changes on a branch, you can merge those changes back into the `main` branch (or any other branch) using the `git merge` command.
- `git merge` takes two commit pointers, usually the branch tips, and will find a **common base commit between them**.
- `git merge` يعنى كوماند دايما بيحاول يدور على الcommon base commit اللى الاتنين branches اللى عاوز اعملهم merge طلعوا منه ومشتركين فيه عشان يبدء من عنده يقارن ويعمل الmerging.
- Once Git finds a common base commit, it will create a new "merge commit" that combines the changes of each queued merge commit sequence.



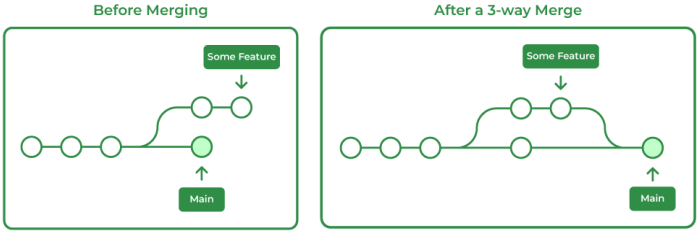


Types of Merges in Git

- **Fast-Forward Merge:**
  - Occurs when the branch being merged has no new commits diverging from the target branch.
  - Git simply moves the HEAD pointer of the target branch forward to match the source branch.
  - The source branch (e.g., `feature`) is based directly on the target branch (e.g., `main`), with no new commits added to the target branch since the source branch was created.
  - Pros
    - Keeps the commit history clean and linear.
    - **No additional merge commit is created**.
  - Cons
    - Doesn't preserve the branch structure, which can make it harder to identify the branching history.



- **Three-Way Merge:**
  - Happens when the branches have diverged (i.e., there are new commits in both branches).
  - Git uses the common ancestor of both branches to combine changes.
  - It uses the differences between the ancestor and each branch to generate a merge commit that combines changes from both.
  - Pros
    - Preserves the branching structure.
    - Useful when combining changes from multiple contributors or resolving diverging histories.
  - Cons
    - Creates a new merge commit, which can clutter the history if not managed well.



Feature	Fast-Forward Merge	Three-Way Merge
Prerequisite	No divergent commits.	Divergent commits exist.
New Merge Commit	No	Yes
Preserves History	Linear history only.	Shows explicit branch structure.
Complexity	Simple.	Requires conflict resolution if needed.
Use Case	Small, sequential changes.	Divergent development paths.

Handling Merge Conflicts in a Three-Way Merge

- If there are changes to the same lines of code in both branches, Git raises a **merge conflict**.
- You must resolve the conflict **manually** before completing the merge.
- **Steps to Resolve a Conflict**
    - Attempt the Merge:

```
git merge feature
```

    - View Conflicts:
        - Conflicted files will be marked, and Git will notify you.
    - Edit the Conflicted Files:
        - Open the files and resolve conflicts by choosing or combining changes.
        - The conflict markers will look like this:

```
<<<<<<< HEAD
Code from the current branch
=======
Code from the feature branch
>>>>>>> feature
```

    - Stage Resolved Files:

```
git add <conflicted-file>
```

    - Complete the Merge:

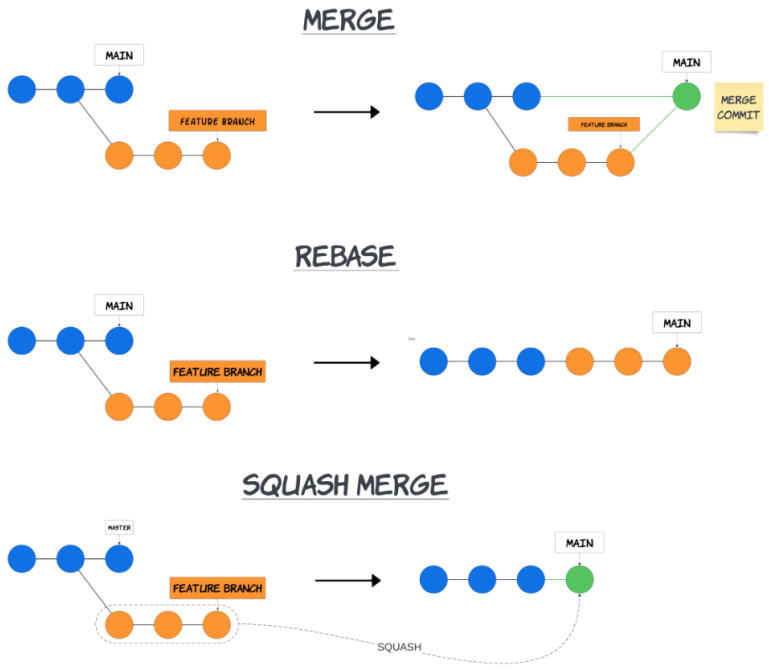
```
git commit
```

- لما جيت عملت برانش جديد وغيرت فيه حاجات عن الماستر ، وبعدين غيرت في الماستر بعيد عن البرانش دة ، وجيت بقى تانى اعمل merge للبرانش دة جوة الماستر ، لاقيت ان في merge conflict ومرضاش يحصل merge ، ولما عملت git status قالى ان الملفات اللى انضافت جديدة مثلا ومكنتش في الماستر اتعملها staging عادى وكانت جاهزة للcommit ، اما الملفات اللى كان فيها conflict كانت لسه unmerged ، وساعتها بقى لازم احل الconflict دة بايدى يعنى افتح الملف اللى فيه المشكلة مثلا بVS code واعمل manual merge للملفات.

```
RTX@Mohaned-PC MINGW64 /d/myGithub/w3schoolRepo (master)
$ git merge hello-world-image
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

RTX@Mohaned-PC MINGW64 /d/myGithub/w3schoolRepo (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   hello.png
        new file:   yes.jpg

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   index.html
```

Squashing a Merge

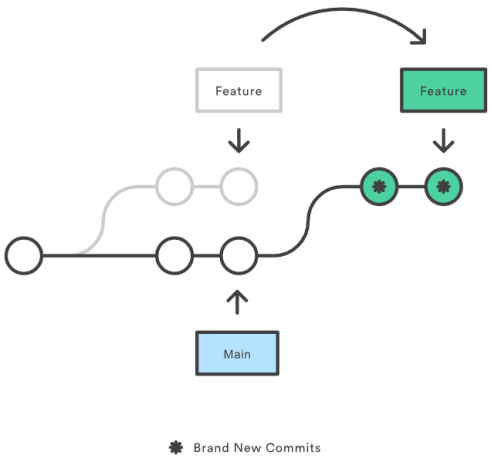
- Squashing a merge combines all the commits from a feature or topic branch into a single commit when merging it into another branch.



Aspect	Regular Merge	Squash Merge
History	Keeps all commits and creates a merge commit.	Combines all commits into one, creating a cleaner history.
Merge Commit	Yes.	Yes (single commit for the merge).
Detail Retention	Retains all individual commit details.	Removes granular commit details.
When to Use	When commit history is important.	When the branch has numerous intermediate commits.

Rebasing

- Rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit.
- It rewrites the commit history by applying commits from the source branch onto the target branch, one by one.
- Internally, Git accomplishes this by creating new commits and applying them to the specified base.
- Even though the branch looks the same, it's composed of **entirely new commits**.
- It maintains a linear history by moving the base of the feature branch.



Feature	Merge	Rebase
Purpose	Combines changes while preserving branch structure.	Combines changes to create a linear history.
New Commit	Creates a merge commit (for three-way merges).	Does not create a merge commit; rewrites history.
Preserves History	Yes, keeps the original branching history.	No, rewrites the history.
Collaboration	Better for team workflows.	Risky if the branch is shared.
Conflict Resolution	Once per merge.	May need to resolve multiple conflicts.
Command	`git merge <branch>`	`git rebase <branch>`
History Appearance	Branching and merging structure visible.	Linear, clean history.

`git branch`	Manage branches in a Git repository. `git branch` ----> List all current branches in local repo. `git branch -a` ----> List all local and remote branches. `git branch -r` ----> List remote branches only. `git branch newBranch` ----> Create a branch. `git branch -d b_Name` ----> Delete a branch safely (check for unmerged changes first) `git branch -D branchName` ----> Force delete a branch. `git branch --track <branch-name> <remote>/<branch-name>` ----> Create a local branch tracking a remote branch. `git branch -m newName` ----> Rename opened branch. `git branch --merged` ----> List all branches that have been merged into the current branch. `git branch -v` ----> Displays the last commit for each branch.
`git switch`	Switch between branches. Create new branches and optionally switch to them. Set up tracking for remote branches. `git switch <branch-name>` ----> Switch to existing branch. `git switch -c <new-branch>` ----> Create a new branch from current HEAD and switch to it. `git switch -` ----> Switch Back to the Previous Branch. `git switch --detach <commit-hash>` ----> Switch to a specific commit in a detached HEAD state. `git switch --track origin/feature-1` ----> Creates a local branch `feature-1` that tracks the remote branch `origin/feature-1`
`git checkout`	Switch between branches. Restore files to a previous state from a specific commit or branch. Create new branches (optionally switching to them). This command is gradually being replaced by more specific commands like `git switch` (for branch management) and `git restore` (for file restoration), but `git checkout` is still widely used and supported. `git checkout newBranch` ----> Switch to existing branch. `git checkout -b newBranch` ----> Create a new branch from current HEAD and switch to it. `git checkout -b newBr Br1` ----> Create a new branch from Br1 and switch to it. `git checkout <commit-hash>` ----> Moves HEAD to a specific commit, placing you in "detached HEAD" mode, where no branch is checked out.
`git merge`	Combine the changes from one branch into another. `git merge <branch_name>` ----> Combines the `<branch_name>` into the current branch.

	Options: • `--no-ff` Forces a merge commit, even if a fast-forward merge is possible. • `--ff-only` Ensures the merge is fast-forward; fails if not possible. • `--abort` Cancels the merge and reverts to the pre-merge state. • `--squash` Combines all commits from the source branch into a single commit. • `--continue` Completes a merge after resolving conflicts.
`git rebase`	Reapply commits from one branch on top of another, effectively rewriting commit history. `git rebase <base_branch>` ----> Moves the commits of the current branch to the tip of `<base_branch>.` **Options:** • `-i` Interactive rebase for editing commits. • `--continue` Continue rebasing after resolving conflicts. • `--abort` Abort the rebase process and return to the original branch state. • `--skip` Skip the current conflicting commit and continue rebasing.

Clone

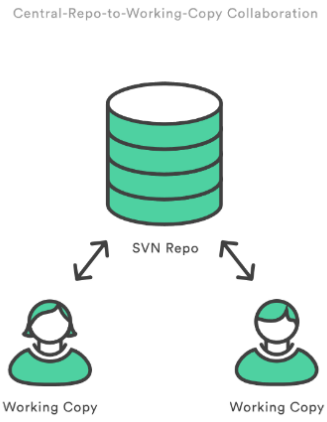
- If you want to have a copy of an existing Git repository, you use the `git clone` command.
- This creates a new directory on your machine with all the repository's files and history.
- The original repository can be located **on the local filesystem** or **on remote machine**.
- The "working copy" is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

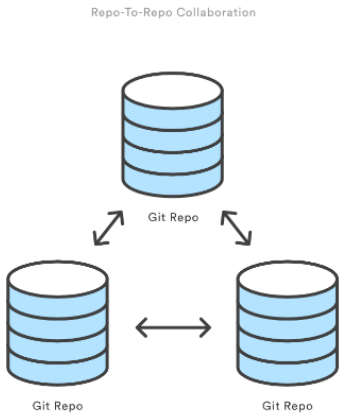
- في فرق بين الcloning والdownloading ، الclone معناها انى باخد نسخة من remote repo وتبقى local عندى على الجهاز وجواها `.git/` فيه كل الhistory بتاع البروجيكت ، لكن download مجرد انى بنزل ملفات المشروع عندى على الجهاز في ملف zip. مثلا ومش بيكون جواه اى معلومات عن الgit history.
- ممكن الcloning يحصل من repo موجودة عندى على الجهاز برضو ، مش لازم يبقى الclone بس لremote repo.

- Cloning automatically creates a remote connection called "`origin`" pointing back to the original repository.
- This automatic connection is established by creating Git refs to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.

Repo-to-Repo Collaboration

- It's important to understand that Git's idea of a "working copy" is very different from the working copy you get by checking out code from an SVN repository.
- SVN stands for Subversion, it is an open-source tool for centralized version control system.
- Unlike SVN, Git makes no distinction between the working copy and the central repository—they're all full-fledged Git repositories.
- Whereas SVN depends on the relationship between the central repository and the working copy, Git's collaboration model is based on repository-to-repository interaction.
- Instead of checking a working copy into SVN's central repository, you push or pull commits from one repository to another.





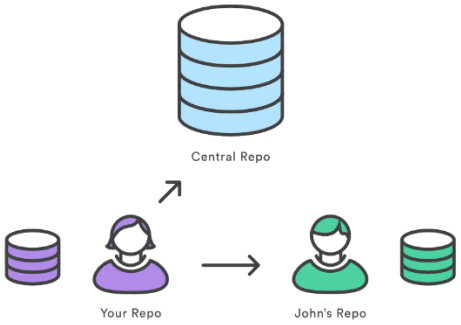
Git URL protocols

- **SSH:** Because SSH is an authenticated protocol, you'll need to establish credentials with the hosting server before connecting. `ssh://[user@]host.xz[:port]/path/to/repo.git/`
- **GIT:** A protocol unique to git. Git comes with a daemon that runs on port (9418). The protocol is similar to SSH however it has NO AUTHENTICATION. `git://host.xz[:port]/path/to/repo.git/`
- **HTTP/S:** The protocol of the web, most commonly used for transferring web page HTML data over the Internet. `http[s]://host.xz[:port]/path/to/repo.git/`

`git clone`	Create a copy of a remote repository on your local machine. `git clone <repo> <local_directory>` `git clone --branch <tag> <repo>` ----> Clone a specific tag. `git clone -b <branch_name> <repository_url>` ----> Clone a specific branch `git clone -depth=3 <repo>` ----> Only clone the commits specified by depth (Shallow clone).

Remote Repositories

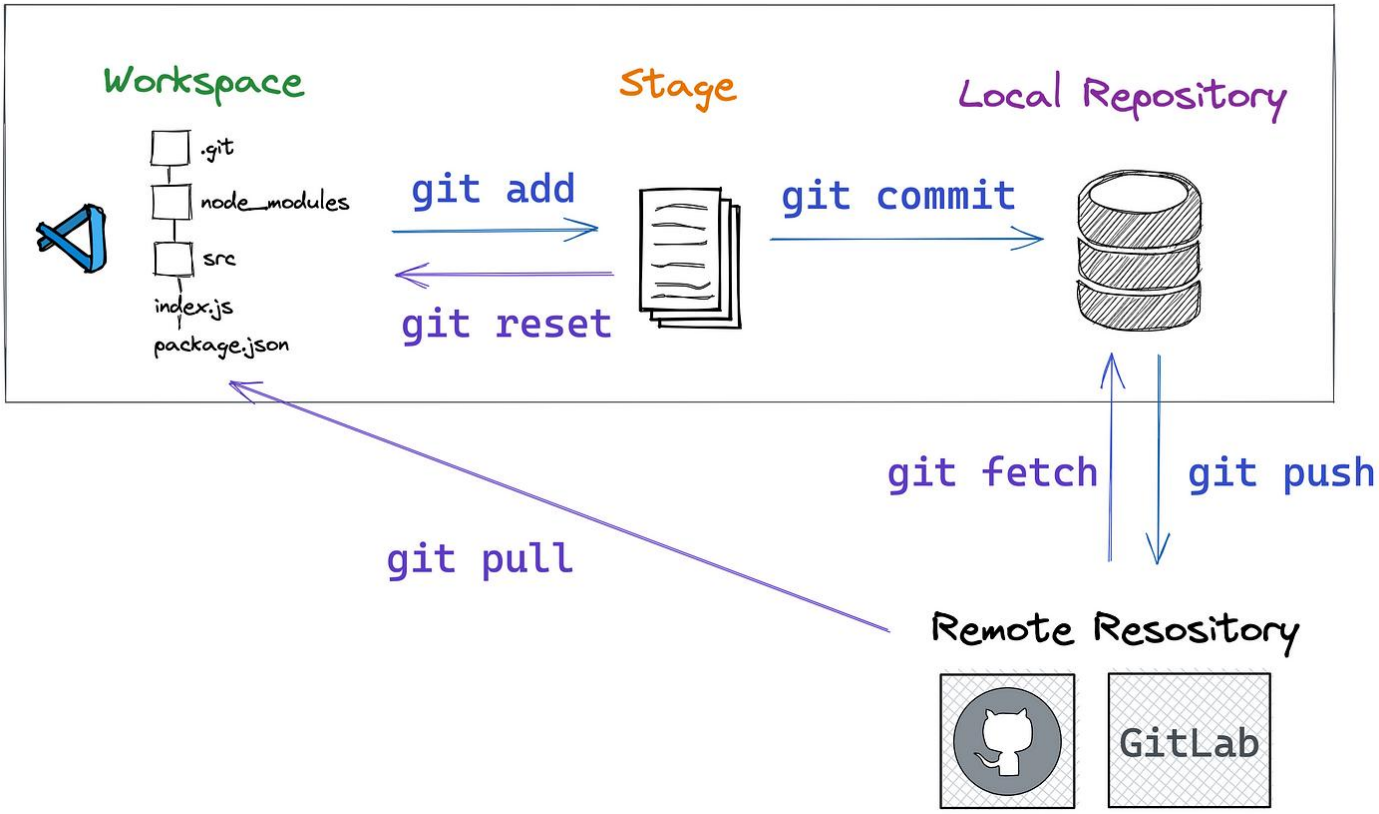
- While you work locally on your machine, Git also allows you to connect to remote repositories using the `git remote` command.
- ممكن أكون عامل fork على github لRepo معين ، fork يعني واخد منه نسخة على الاكونت عندى ، بعد كدة عملت clone للـ forked repo دة عندى locally ، ساعتها بقى اقدر اضيف remote تانى للـ local repo دة والريموت دة يكون الـrepo الاصلى عشان مثلا اى تعديلات جديدة تحصل في الـrepo الاصلى اقدر اعملها fetch او pull عندى locally وبعدين ابقى اعمل push للـforked repo اللى على الأكونت عندى.
- The `git remote` command lets you create, view, and delete connections to other repositories.
- Remote connections are more like bookmarks rather than direct links into other repositories.
- Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.
- For example, the following diagram shows two remote connections from your repo into the central repo and another developer's repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



`git remote`	An interface for managing a list of remote entries that are stored in the repository's ./.git/config file.	
	`git remote`	----> Lists all configured remote repositories.
	`git remote -v`	----> Include the URL of each connection.
	`git remote add <name> <url>`	----> Create a new connection to a remote repository.
	`git remote rm <name>`	----> Remove a configured remote repository.
	`git remote set-url <name> <new_url>`	----> Updates the URL of an existing remote.
	`git remote rename <old-name> <new-name>`	----> Renames an existing remote repository.

Push and Pull

Local

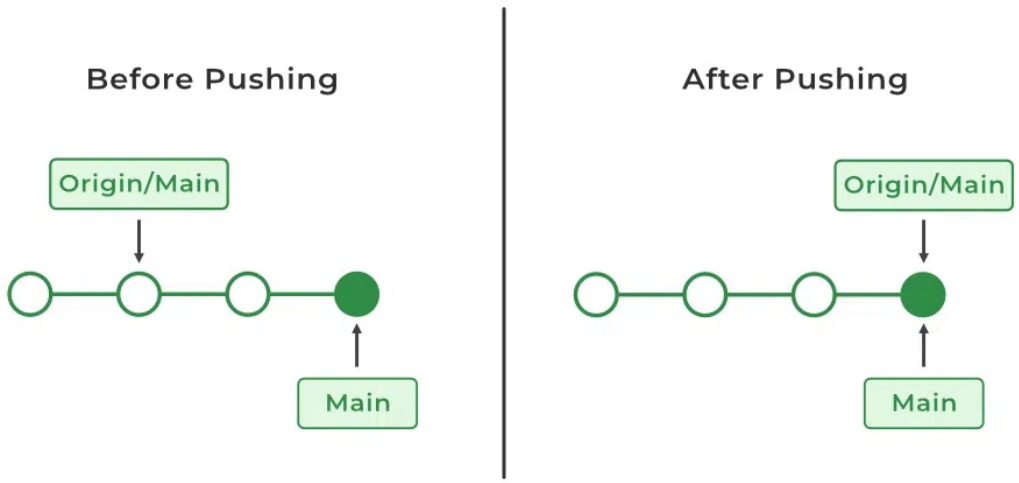


Pushing

- Pushing is the process of sending your local commits (changes) to a remote repository.
- This makes your changes available to others working on the project.
- **Example**

```
git push <remote> <branch>
git push origin main
```

  - `origin`: The remote repository (typically the default remote).
  - `main`: The branch you are pushing to.



How Git Push Works

- Git checks your local branch to ensure it is up-to-date with the remote branch.
- If there are no conflicts, your local commits are sent to the remote repository.
- If conflicts exist (e.g., the remote has new commits you don't have locally), the push is rejected. You'll need to pull and resolve the conflicts first.

Fetching

- Fetching in Git refers to downloading the latest changes (commits, branches, tags) from a remote repository to your local repository without integrating them into your working branch.
- It is a safe way to inspect updates and changes made on the remote before deciding how to apply them.
- **Syntax**

```
git fetch <remote> <branch>
```

  - `<remote>`: The name of the remote repository (e.g., `origin`).
  - `<branch>`: The specific branch you want to fetch. Optional.

How Git Fetch Works

- Retrieves new commits, tags, and branches from the remote repository.
- Updates your local references (e.g., `origin/main`) without modifying your working directory.
- Leaves your current branch and files unchanged.

Workflow with git fetch

- Fetch Remote Changes:

```
git fetch origin
```

- Check the Status of Your Branch:

```
git status
```

  - Git will show if your branch is ahead or behind the remote branch.

- Review Fetched Changes:
  - Compare your branch with the fetched branch:

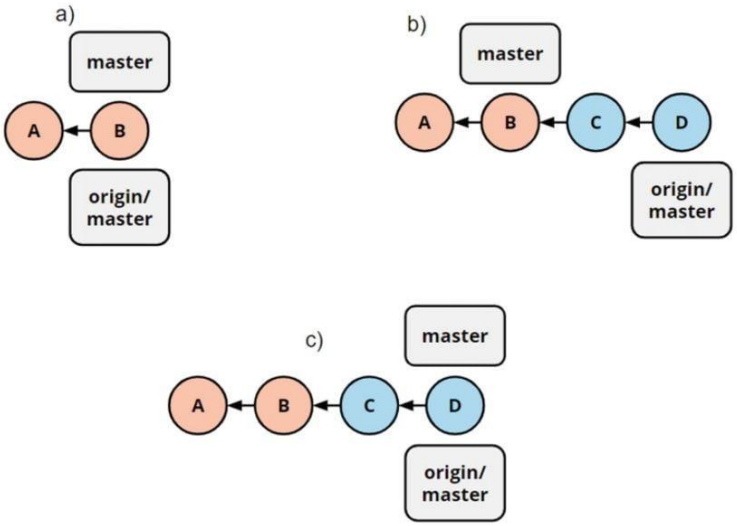
```
git diff origin/main
```

- Decide How to Apply Changes:
  - Merge:

```
git merge origin/main
```

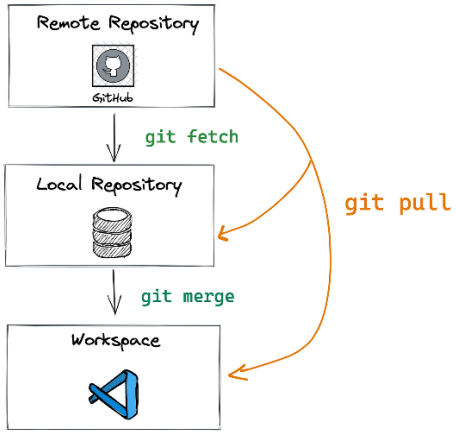
  - Rebase:

```
git rebase origin/main
```



Pulling

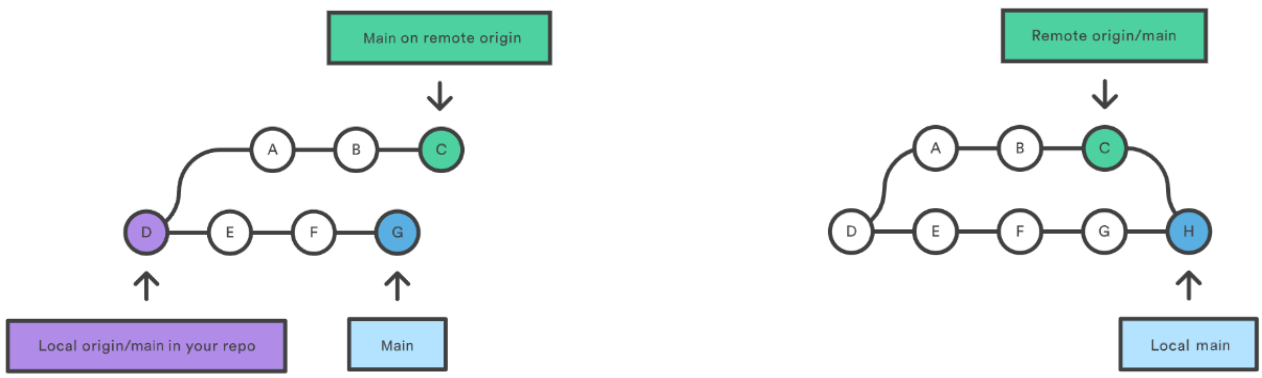
- Pulling is the process of fetching changes from a remote repository and merging them into your current branch.
- It's a combination of **<u>two</u>** Git commands:
  - `git fetch`: Downloads changes from the remote repository without applying them.
  - `git merge`: Integrates the fetched changes into your branch.



- **Example**

```
git pull <remote> <branch>
git pull origin main
```

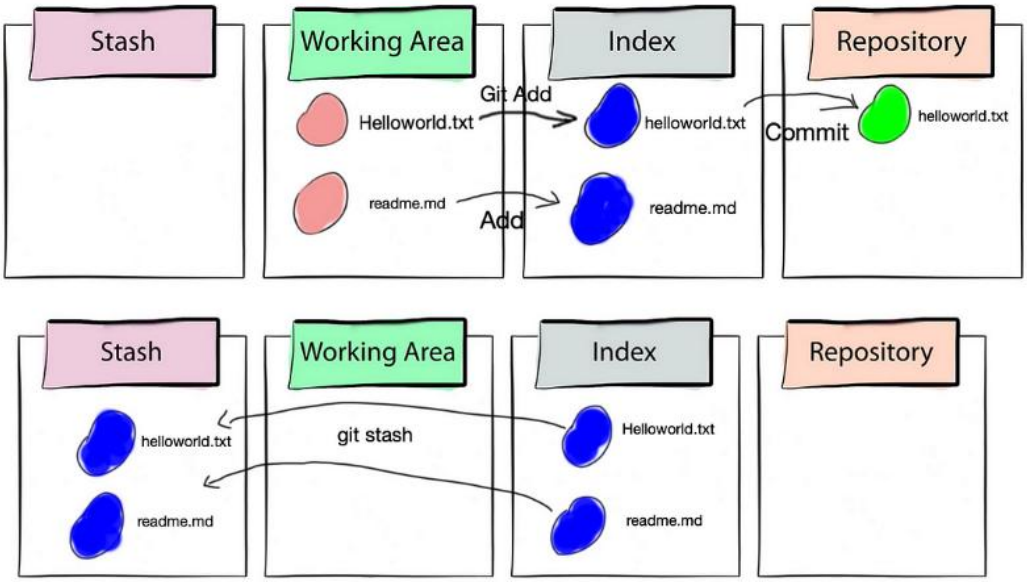
  - `origin`: The remote repository.
  - `main`: The branch you want to pull changes from.



`git push`	It is used to upload local repository content to a remote repository. `git push <remote> <branch>` ----> Push the specified branch to remote. `git push <remote> --force` ----> Force the push even if it results in a non-fast-forward merge. `git push <remote> --tags` ----> Push All Tags. `git push origin <tag-name>` ----> Push a Specific Tag. `git push origin --delete <branch-name>` ----> Delete a branch from remote repo. `git push origin --delete <tag-name>` ----> Delete a Remote Tag.
`git fetch`	Download the latest changes (commits, files, and branches) from a remote repository but does not automatically merge them into your working branch. `git fetch <remote>` ----> Fetch all of the branches from the repository. `git fetch <remote> <branch>` ----> Only fetch the specified branch. `git fetch --tags` ----> Fetch all tags from the remote. `git fetch --all` ----> Fetch from all configured remotes.
`git pull`	It is used to pull all changes from a remote repository into the branch you are working on. `git pull <remote>` ----> Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. **Options:** • `--rebase` Use rebase instead of merge when applying changes from the remote branch. • `--ff-only` Prevent merge commits if your local branch cannot be fast-forwarded. • `--no-commit` Fetch and merge but stop before committing.

Stashing

- Stashing in Git is a way to temporarily save uncommitted changes in your working directory (and optionally staged changes) without committing them.
- This is useful when you need to switch branches or work on something else without losing or committing your current work.
- It saves your uncommitted changes in a stack-like structure.
- Stashes are stored locally and cannot be pushed to a remote repository.
- Untracked and ignored files are not included by default.



Scenarios for Using Stashing

- **Switching Branches with Uncommitted Changes:**
  - If you have uncommitted changes and need to switch to another branch:

```
git stash
git switch <branch>
```

  - After completing your work, reapply the stash:

```
git stash pop
```

- **Temporary Pause in Development:**
  - When you need to test or debug another part of the codebase:

```
git stash
```

  - Resume later:

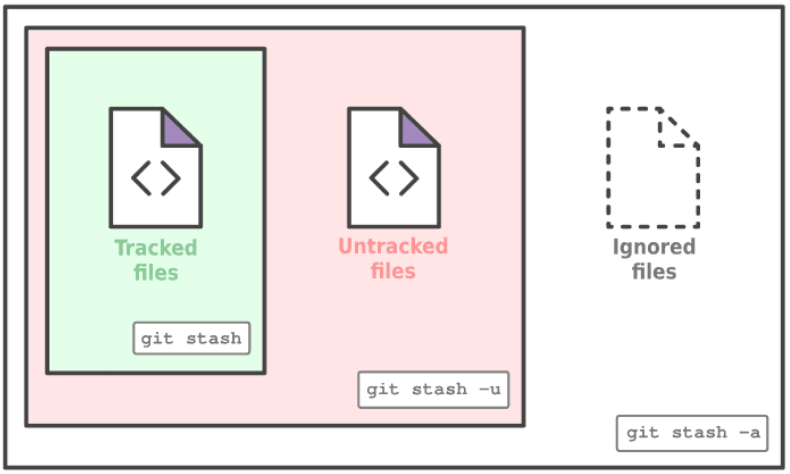
```
git stash apply
```

- **Cleaning the Working Directory Temporarily:**
  - To get a clean state without committing changes:

```
git stash -u
```

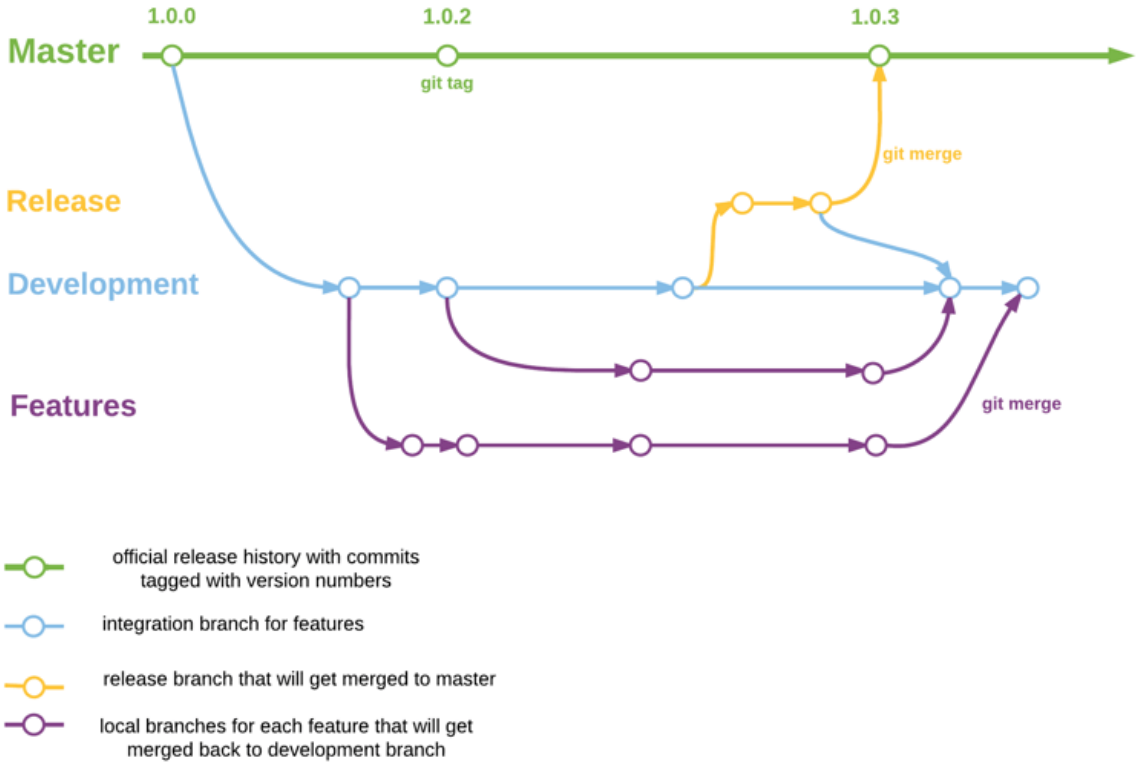
  - After completing your work, restore untracked files as well:

```
git stash pop
```

`git stash`	Temporarily save (or "stash") changes in your working directory and staging area without committing them. `git stash` ----> Stashing any tracked files (staged, unstaged) that are not committed yet. `git stash -u` ----> Includes untracked files. `git stash -a` ----> Includes ignored files. Stash Everything (Tracked, Untracked, and Ignored Files)  `git stash save "message on stashes list"` ----> Stashing with customized message. `git stash list` ----> list all stashes IDs. `git stash pop` ----> Un-stashing latest added files. `git stash pop stash@{2}` ----> Un-stashing stash number 2. `git stash apply` ----> Un-stashing a copy of latest added files. `git stash apply stash@{2}` ----> Un-stashing a copy of stash number 2. `git stash drop` ----> Dropping the latest stash with its content. `git stash drop stash@{2}` ----> Dropping stash number 2 with its content. `git stash show` ----> Show the content of latest stash. `git stash show stash@{2}` ----> Show the content of stash number 2. `git stash branch BranchName stash@{2}` ----> Change to `BranchName` branch and un-stashing stash number 2 in it. `git stash clear` ----> Deleting all stashes with their contents.

Tagging

- Tagging in Git is a way to mark specific points in a repository's history, usually for important milestones like releases or versions.
- Tags are often used to indicate software release versions (e.g., v1.0.0) and are helpful when you want to easily reference a specific commit.



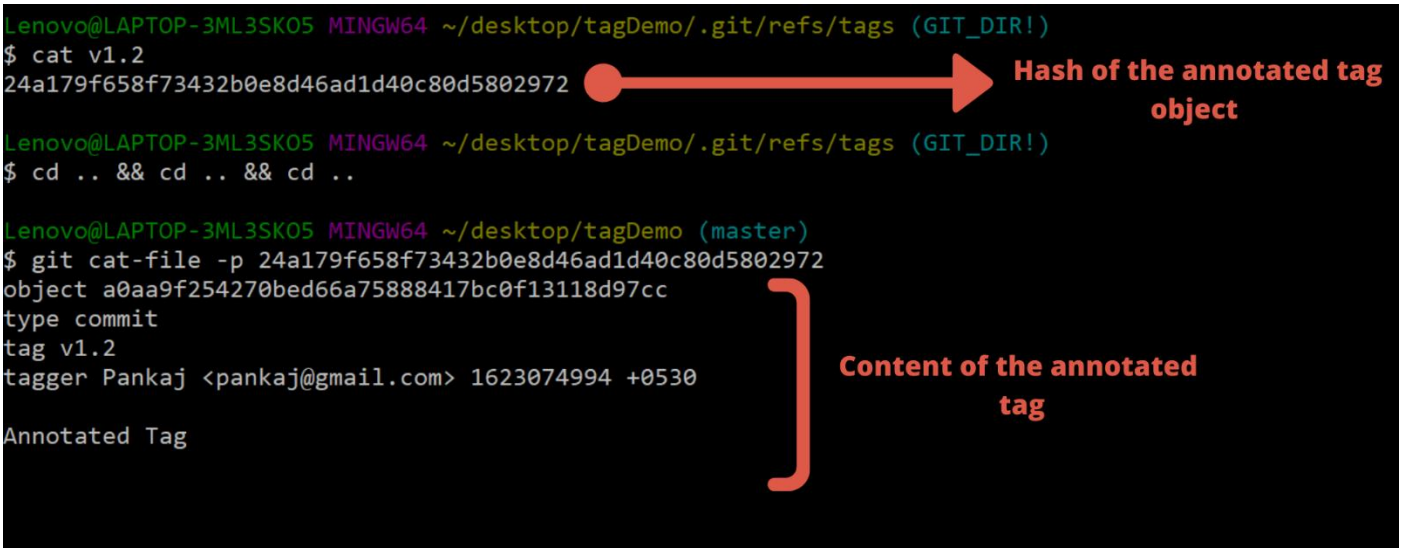
Types of Git Tags

- **Lightweight Tags:**
  - Lightweight or Unannotated tags are just **pointers to specific commit**.
  - A lightweight tag just stores the hash of the commit it points to and no other information.
  - They are mainly used on local systems only and it is not recommended to push them to the remote repository as they do not add much value.
  - Best used for temporary markers or quick references.
  - We can view the tags by navigating to the `.git/refs/tags` directory.

```
Lenovo@LAPTOP-3ML3SKO5 MINGW64 ~/desktop/tagDemo (master)
$ cd .git/refs/tags

Lenovo@LAPTOP-3ML3SKO5 MINGW64 ~/desktop/tagDemo/.git/refs/tags (GIT_DIR!)
$ cat v1.1
a0aa9f254270bed66a75888417bc0f13118d97cc

Lenovo@LAPTOP-3ML3SKO5 MINGW64 ~/desktop/tagDemo/.git/refs/tags (GIT_DIR!)
$ cd .. && cd .. && cd ..

Lenovo@LAPTOP-3ML3SKO5 MINGW64 ~/desktop/tagDemo (master)
$ git log
commit a0aa9f254270bed66a75888417bc0f13118d97cc (HEAD -> master, tag: v1.2, tag: v1.1)
Author: Pankaj <pankaj@gmail.com>
Date:   Mon Jun 7 19:39:41 2021 +0530

    Initial Commit
```

Hash of the object that the lightweight tag points to

- **Annotated Tags:**
  - Annotated tags are tags that contain additional information and not just the hash of the commit.
  - They are **full objects in the Git database**.
  - The additional information (called metadata) can have fields like the type of object the tag points to, the name and email of the person who created the tag, a tag message.
  - They even have their own hash as they are objects and not just pointers.
  - Annotated tags should be used if the branch is to be pushed on a remote repository.
  - To view all this information we first need to get the hash of the tag from the `.git/refs/tags` directory.
  - Cryptographically signed if desired.
  - Recommended for marking official releases.



Use Cases for Tagging

- **Releases**: Mark a commit as a specific release version.
- **Milestones**: Identify important points in the repository's history (e.g., "v1.0.0").
- **Debugging**: Reference a specific state for testing or debugging.

`git tag`	Create, list, delete, and manage tags in Git. `git tag [options] <tag_name> [<commit>]` • `<tag_name>`: The name of the tag. • `<commit>`: Optional; the commit to tag. **Defaults** to the latest commit on the current branch. `git tag` ----> Display all tags in the repository. `git tag -l "v1.*"` ----> Filter tags matching a pattern. `git tag v1.0.0` ----> Create a Lightweight Tag. `git tag -a <tag_name> -m "Tag message"` ----> Create an Annotated Tag. `git tag -s <tag_name> -m "Tag message"` ----> Create a Signed Tag. `git tag <tag_name> <commit_hash>` ----> Create a Tag for a Specific Commit. `git show <tag_name>` ----> Display details of an annotated tag. `git push origin <tag_name>` ----> Push a single tag. `git push origin --tags` ----> Push all tags. `git tag -d <tag_name>` ----> Delete a tag locally. `git push origin --delete <tag_name>` ----> Delete a tag from a remote repository.

Git ignore

- Git sees every file in your working directory as one of **three states**:
  - **tracked** a file which has been previously staged or committed.
  - **untracked** a file which has not been staged or committed.
  - **ignored** a file which Git has been explicitly told to ignore.
- Ignored files are usually built-artifacts and machine-generated files that can be derived from your repository source or should otherwise not be committed.
- Some common examples are:
  - Dependency caches, such as the contents of `/node_modules` or `/packages`
  - Compiled code, such as `.o`, `.pyc`, and `.class` files
  - Build output directories, such as `/bin`, `/out`, or `/target`
  - Files generated at runtime, such as `.log`, `.lock`, or `.tmp`
  - Hidden system files, such as `.DS_Store` or `Thumbs.db`
  - Personal IDE config files, such as `.idea/workspace.xml`
- Ignored files are tracked in a special file named `.gitignore` that is checked in at the root of your repository.
- There is **no explicit git ignore command**, instead, the `.gitignore` file must be edited and committed by hand when you have new files that you wish to ignore.
- `.gitignore` files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.
- **Scope:**
  - `.gitignore` rules only apply to untracked files.
  - If a file is already tracked by Git (committed to the repository), `.gitignore` won't ignore it unless the file is removed from tracking.
- **Global vs Local:**
  - A project-specific `.gitignore` resides in the root of the repository.
  - A global `.gitignore` applies rules across all repositories for a user.
- Use `.gitignore` templates for common programming languages.
  - GitHub provides starter `.gitignore` templates:

https://github.com/github/gitignore

Setting Up a Global .gitignore File

- **Create the Global .gitignore File**
  - You can create the file in your home directory or any preferred location.

```
touch ~/.gitignore_global
```

- **Add Rules to the File**
  - Edit the `.gitignore_global` file and add patterns for files or directories you want to ignore globally.
  - Example:

```
# macOS system files
.DS_Store

# Editor settings
.vscode/
.idea/
*.swp

# Log files
*.log
```

- **Configure Git to Use the Global .gitignore File**
    - Run the following command to tell Git to use your global .gitignore file:

```
git config --global core.excludesfile ~/.gitignore_global
```

    - `core.excludesfile`: This configuration key tells Git the location of the global `.gitignore` file.
    - Replace `~/.gitignore_global` with the full path if your file is stored elsewhere.
- **Verify the Configuration**
    - To check if Git is using the global `.gitignore` file:

```
git config --get core.excludesfile
```

    - It should display the path to your global `.gitignore` file.

Committing an ignored file

- By default, Git ignores files that match the patterns in `.gitignore`
- However, there might be scenarios where you want to commit an ignored file intentionally (e.g., adding a temporary file for debugging purposes or committing sensitive files for later removal).
- Steps to Commit an Ignored File
    - **Verify the Ignored File Check if the file is being ignored by Git:**

```
git check-ignore -v <file>
```

        - If it's ignored, Git will show the rule and the .gitignore file responsible.
    - **Force Add the Ignored File Use the `-f` or `--force` option with the `git add` command to override .gitignore rules:**

```
git add -f <file>
```

        - Example:

```
git add -f config/settings.json
```

    - **Commit the File After adding the file, commit it as usual:**

```
git commit -m "Force commit an ignored file"
```

- However, a better solution is to define an exception to the general rule:

```
$ echo !debug.log >> .gitignore
$ cat .gitignore
*.log
!debug.log
$ git add debug.log
$ git commit -m "Adding debug.log"
```

- This approach is more obvious, and less confusing, for your teammates.

`git check-ignore`	Identify whether a specific file or directory is being ignored by Git based on .gitignore rules and other ignore mechanisms.
	
	`git check-ignore [options] <file>...`
	
	Options:
	• `-v`, `--verbose` Show the ignore rule and its origin (e.g., .gitignore or global ignore).
	• `-z` Output paths with a null character (for scripting).
	
	`git check-ignore debug.log` ----> Checking a Single File.
	`git check-ignore debug.log temp/data.txt` ----> Checking Multiple Files.
	`git check-ignore -v <file>` ----> See the exact ignore rule and the `.gitignore` file responsible.
	
	Example Output:
	
	`.gitignore:3:*.log debug.log`
	
	• `.gitignore` is the file where the rule is defined.
	• `3` indicates the line number of the rule in .gitignore.
	• `*.log` is the ignore pattern applied.

Git ignore patterns

- `.gitignore` uses globing patterns to match against file names.
- You can construct your patterns using various symbols:

Pattern	Description
`*.log`	Ignore all files with `.log` extension.
`temp/`	Ignore a directory named `temp` and its contents.
`/debug.log`	Ignore only the `debug.log` file in the root directory.
`!important.log`	Do not ignore `important.log`, even if `*.log` is ignored.
`*.log` `!important.log`	Ignores all `.log` files but explicitly excludes `important.log`
`config/*.json`	Ignore all `.json` files in the `config` directory.
`**/logs/*.log`	Ignore all `.log` files in any `logs` directory at any level.
`# comment`	Lines starting with `#` are comments.

Special Characters in Git Ignore Patterns

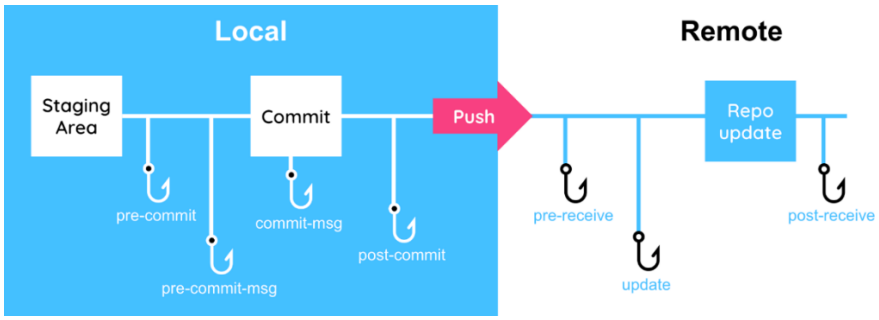
Character	Description
`*`	Matches zero or more characters.
`?`	Matches any single character.
`[]`	Matches one of the characters inside the brackets (e.g., [abc] matches a, b, or c).
`!`	Negates the pattern to include the file explicitly.
`/`	Anchors the pattern to a directory or file in the root.
`#`	Starts a comment line.

Git Hooks

- Git hooks are scripts that are executed automatically by Git at specific events during the Git workflow.
- They allow you to customize and automate parts of your Git process, such as enforcing coding standards, running tests, or deploying code after a push.

Types of Git Hooks

- Git hooks are categorized into two types:
  - **Client-Side Hooks**
    - Triggered by operations such as committing, merging, and checking out code.
    - Common examples:
      - `pre-commit`
      - `prepare-commit-msg`
      - `commit-msg`
      - `post-commit`
  - **Server-Side Hooks**
    - Triggered by events on a Git server, like receiving pushed commits.
    - Common examples:
      - `pre-receive`
      - `update`
      - `post-receive`



Hook File Structure

- Git hooks are stored in the `.git/hooks` directory of a repository.
- Each hook is a file with a specific name (e.g., `pre-commit`, `post-commit`).
- Hooks must be executable scripts (e.g., shell scripts, Python, or any language supported by your system).

Common Hooks and Their Uses

Hook Name	Type	When It's Triggered	Use Case
`pre-commit`	Client-Side	Before the commit is created	Code linting, running tests, or checking formatting before committing.
`prepare-commit-msg`	Client-Side	Before the commit message editor is opened	Modifying or auto-generating commit messages.
`commit-msg`	Client-Side	After the commit message is entered	Validating the commit message format.
`post-commit`	Client-Side	After a commit is created	Logging, notifications, or further automated tasks.
`pre-push`	Client-Side	Before git push sends data to the remote repository	Running tests or checking branch policies.
`pre-receive`	Server-Side	Before updating a remote repository with pushed commits	Validating pushed changes or enforcing access control.
`update`	Server-Side	When a branch or tag is updated on the server	Enforcing branch protection rules.
`post-receive`	Server-Side	After a push is completed	Triggering CI/CD pipelines or deployment scripts.

How to Set Up Git Hooks

- **Navigate to the .git/hooks Directory**
  - Each Git repository has a `.git/hooks` directory containing sample hooks (e.g., `pre-commit.sample`).
- **Create or Modify a Hook File**
  - To install a hook, remove the `.sample` extension to make it executable.
  - Example: `pre-commit` hook (Without `.sample` extension)

```
#!/bin/sh
echo "Running pre-commit hook..."
# Run linting
eslint .
# Abort commit if linting fails
if [ $? -ne 0 ]; then
    echo "Linting failed. Commit aborted."
    exit 1
fi
```

- **Make the Hook Executable**

```
chmod +x .git/hooks/pre-commit
```

- **Test the Hook**
  - Perform the corresponding Git operation (e.g., `git commit`) to trigger the hook.

Global Git Hooks

- While hooks are repository-specific by default, you can define global hooks using a custom template directory.
  - **Set Up a Global Template Directory**

```
mkdir ~/.git-templates/hooks
```

  - **Add Hook Scripts to the Directory Example:** Create a pre-commit script in `~/.git-templates/hooks`.
  - **Configure Git to Use the Template Directory**

```
git config --global init.templateDir ~/.git-templates
```

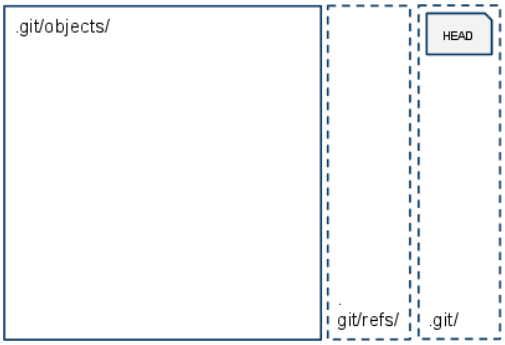
  - **Apply the Template to Existing Repositories**
    - Reinitialize the repository:

```
git init
```

Explanations

Creating a Git repository

(1) Initialize New Repository: `$ git init`

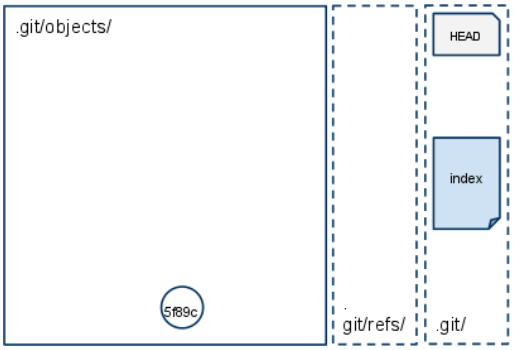


What happened:

- Empty `.git/objects/` and `.git/refs/` created.
- No index file yet.
- `HEAD` symbolic reference created.

```
$ cat .git/HEAD
ref: refs/heads/master
```

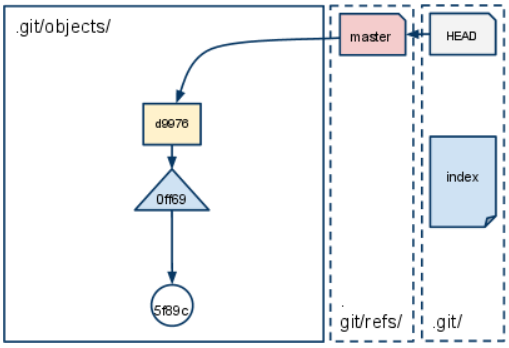
(2) Add a new file and stage it: `$ echo "A roti canai project." >> README`
`$ git add README`



What happened:

- Index file created: It has a SHA1 hash that points to a blob object.
- Blob object created: The content of README file is stored in this blob.

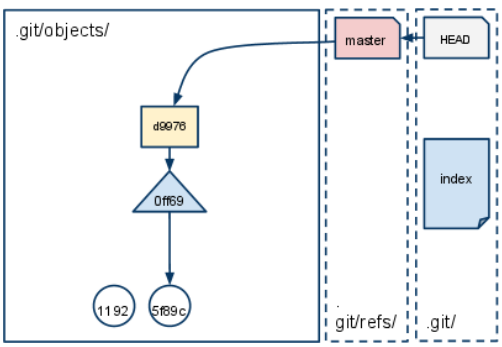
(3) First Commit: `$ git commit -m "first commit"`



What happened:

- Branch 'master' reference created: It points to the latest commit object in 'master' branch.
- First commit object created. It points to the root tree object.
- Tree object created: This tree represents the 'canai' directory.

(4) Add Modified File: `$ echo "Welcome everyone." >> README`
`$ git add README`

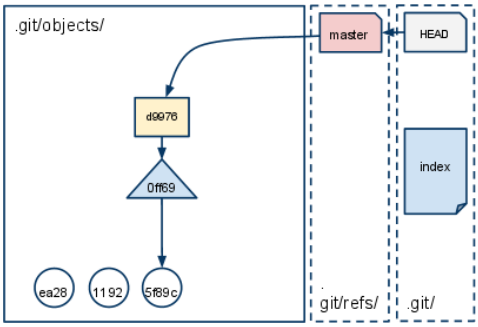


What happened:

- Index file updated.
- Blob object created: The entire README content is stored as a new blob.

(5) Add File into Subdirectory:

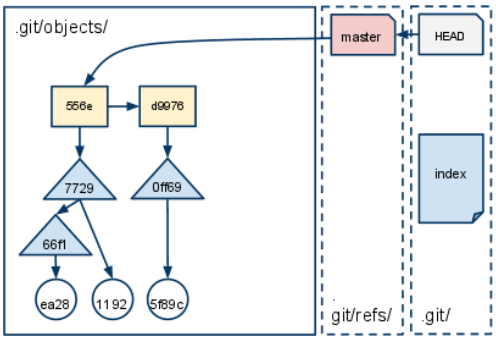
```
$ mkdir doc
$ echo “[[TBD]] manual toc” >> doc/manual.txt
$ git add doc
```



What happened:

- Index file updated.
- Blob object created.

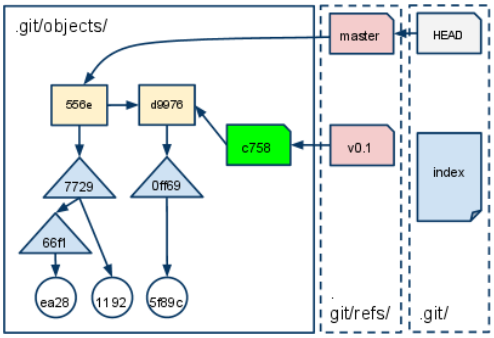
(6) Second Commit: `$ git commit -m ’second commit’`



What happened:

- Branch ‘master’ reference updated: It points to the latest commit in this branch.
- Second commit object created: Notice its ‘parent’ points to the first commit object. This forms a commit graph.
- New root tree object created.
- A new subdir tree object created.

(7) Add Annotated Tag: `$ git tag -a -m ‘this is annotated tag’ v0.1 d9976`



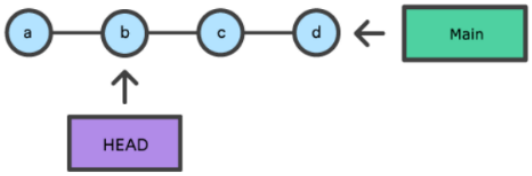
What happened:

- Tag reference created.
- It points to a commit object.

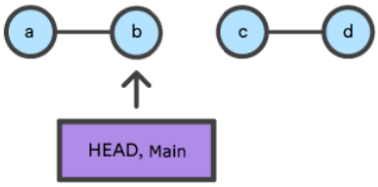
`git reset`

- The `git reset` command is a complex and versatile tool for undoing changes.
- It has three primary forms of invocation corresponding to CLI options `--soft`, `--mixed`, `--hard`.
- The three arguments each correspond to Git's three internal state management mechanism's:
    - The Commit Tree (HEAD).
    - The Staging Index.
    - The Working Directory.
- `git reset` will move the `HEAD` ref pointer and the current branch ref pointer.
- `git reset` will **never delete a commit**, however, commits can become 'orphaned' which means there is no direct path from a ref to access them.
- Git will permanently delete any orphaned commits after it runs the internal garbage collector. By default, Git is configured to run the garbage collector every **30 days**.
- To better demonstrate this behavior, consider the following example:

- This example demonstrates a sequence of commits on the main branch. The `HEAD` ref and `main` branch ref currently point to commit `d`.
- Now let us execute and compare, both `git checkout b` and `git reset b`.



    - With `git checkout`, the `main` ref is still pointing to `d`. The `HEAD` ref has been moved, and now points at commit `b`. The repo is now in a **'detached HEAD'** state.



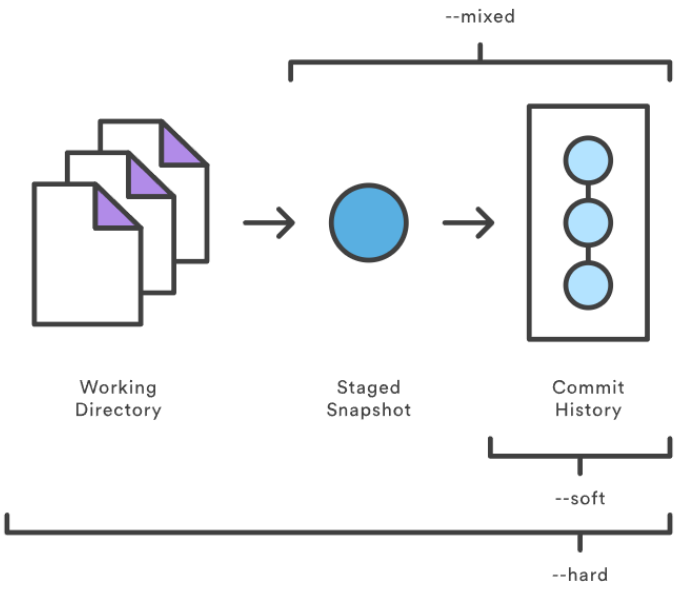
    - Comparatively, `git reset`, moves both the `HEAD` and branch refs to the specified commit.

- The `git reset HEAD~2` command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history.
- Remember that this kind of reset should only be used on unpublished commits. Never perform the above operation if you've already pushed your commits to a shared repository.

Main Options

- The default invocation of `git reset` has implicit arguments of `--mixed` and `HEAD`.
- In this form `HEAD` is the specified commit.
- Instead of `HEAD` any Git SHA-1 commit hash can be used.

The scope of git reset's modes



--hard

- This is the most direct, **DANGEROUS**, and frequently used option.
- Moves the `HEAD` to the specified commit.
- Updates the staging area and working directory to match the target commit.
- Completely discards any changes that were staged or made in the working directory.

--mixed (default): Unstage Changes

- Moves the `HEAD` to the specified commit.
- Updates the staging area to match the target commit.
- Leaves the working directory unchanged.

--soft: Keep Changes in Staging

- Moves the `HEAD` to the specified commit.
- Does not modify the staging area or working directory.
- Changes in the undone commits remain staged.

Practical Scenarios

Fix a Commit History (--soft):

- If you mistakenly made multiple commits and want to combine them:

```
git reset --soft HEAD~2
git commit -m "New combined commit"
```

Unstage Changes (--mixed):

- If you mistakenly staged changes and want to review them first:

```
git reset HEAD
```

Clean Up a Messy Working Directory (--hard):

- If you've made changes that you no longer want:

```
git reset --hard HEAD
```

Feature	--soft	--mixed (default)	--hard
Effect on HEAD	Moves `HEAD` to the target commit.	Moves `HEAD` to the target commit.	Moves `HEAD` to the target commit.
Effect on Staging Area	Leaves staging area unchanged.	Clears the staging area.	Clears the staging area.
Effect on Working Dir	Leaves working directory unchanged.	Leaves working directory unchanged.	Discards all changes in the working directory.
Use Case	Undo commits but keep changes staged.	Undo commits and unstage changes.	Completely discard changes (staged and unstaged).

GitHub Flow

- The GitHub flow is a workflow designed to work well with Git and GitHub.
- It focuses on branching and makes it possible for teams to experiment freely and make deployments regularly.
- The GitHub flow works like this:
    - Create a new Branch
    - Make changes and add Commits
    - Open a Pull Request
    - Review
    - Deploy
    - Merge

Create a New Branch

- Branching is the key concept in Git. And it works around the rule that the master branch is ALWAYS deployable.
- That means, if you want to try something new or experiment, you create a new branch! Branching gives you an environment where you can make changes without affecting the main branch.
- When your new branch is ready, it can be reviewed, discussed, and merged with the main branch when ready.
- When you make a new branch, you will (almost always) want to make it from the master branch.
- **Note:** Keep in mind that you are working with others. Using descriptive names for new branches, so everyone can understand what is happening.

Make Changes and Add Commits

- After the new branch is created, it is time to get to work. Make changes by adding, editing and deleting files. Whenever you reach a small milestone, add the changes to your branch by commit.
- Adding commits keeps track of your work. Each commit should have a message explaining what has changed and why. Each commit becomes a part of the history of the branch, and a point you can revert back to if you need to.
- **Note:** commit messages are very important! Let everyone know what has changed and why. Messages and comments make it so much easier for yourself and other people to keep track of changes.

Open a Pull Request

- Pull requests are a key part of GitHub. A Pull Request notifies people you have changes ready for them to consider or review.
- You can ask others to review your changes or pull your contribution and merge it into their branch.

Review

- When a Pull Request is made, it can be reviewed by whoever has the proper access to the branch. This is where good discussions and review of the changes happen.
- Pull Requests are designed to allow people to work together easily and produce better results together!
- If you receive feedback and continue to improve your changes, you can push your changes with new commits, making further reviews possible.
- **Note:** GitHub shows new commit and feedback in the "unified Pull Request view".

Deploy

- When the pull request has been reviewed and everything looks good, it is time for the final testing. GitHub allows you to deploy from a branch for final testing in production before merging with the master branch.
- If any issues arise, you can undo the changes by deploying the master branch into production again!
- **Note:** Teams often have dedicated testing environments used for deploying branches.

Merge

- After exhaustive testing, you can merge the code into the master branch!
- Pull Requests keep records of changes to your code, and if you commented and named changes well, you can go back and understand why changes and decisions were made.
- **Note:** You can add keywords to your pull request for easier searching!

Best practices

Having develop branch in addition to main branch

يفضل انه يكون عندى develop branch بجانب الmain branch ودة يعتبر intermediary main عشان لو انا مثلا عامل pipeline دى هتكون مربوطة بالmain ولما يحصل اى تعديل في الmain هيحصل trigger ليها وكدة معناه ان كل fixed bug او added feature هتتعمل وهتنضاف للmain هيحصل triggering وبالتالي يفضل انى اجمع الحاجات دى كلها الأول في الdev branch وبعدين ابعتهم مرة واحدة كbig chunk للmain.

Commands

`git status`	Show Status. `git status --short  || git status -s` **Note:** Short status flags are: • `??` - Untracked files • `A` - Files added to stage • `M` - Modified files • `D` - Deleted files
`git help`	`git command -help` ----> See all the available options for the specific command. `git help --all` ----> See all possible commands.
`git init`	Create git repository in current directory. `git init <RepoName>` ----> Create a new repo and initialize .git/ inside it.
`git config`	`git config --list` ----> List all available settings. **Note:** Use `--global` to set the username and e-mail for every repository on your computer. If you want to set the username/e-mail for just the current repo, you can remove global. `git config --global user.name` ----> Return the existing value. `git config --global user.name "Mohaned"` ----> Setting new value. `git config --global --unset user.name ""` ----> Only clearing the config option value. `git config --global --unset user.name` ----> Removing the config option from the list. `git config --global --edit` ----> Editing config file through an editor. `git config --global alias.cm "commit -m"` ----> Aliasing "commit -m" with cm. `git config --global --unset alias.NAME`

`git add`	Add files to staging area.
	
	`git add <File> <File>`
	`git add *.extension`
	`git add *`

`git ls-files`	It shows files that are tracked by Git, i.e., files in the index (staging area) or the working directory that are not ignored. • `--cached` Show files in the index (staged for commit). • `--modified` Show files that are modified in the working directory but not staged. • `--deleted` Show files that are deleted from the working directory but still tracked. • `--others` Show files in the working directory that are not tracked and not ignored. • `--ignored` Show files that are ignored by `.gitignore`. • `--stage` Show files along with their staging information (e.g., file mode, object ID, and stage number).
`git cat-file`	Provide contents or details of repository objects. `git cat-file <type> <object-hash>` ----> Show object content. `git cat-file -t eff6b` ----> Show object type. `git cat-file -s eff6b` ----> Show object size. `git cat-file -p eff6b` ----> Show object content.
`git clean`	Removes untracked files and directories from the working directory. It is helpful for cleaning up a repository by deleting files that are not tracked by Git or ignored by `.gitignore`. By default, Git is globally configured to require that `git clean` be passed a "force" option to initiate. `git clean -n` ----> Show what untracked files would be deleted. `git clean -f` ----> Force deletes untracked files. `git clean -df` ----> Force deletes untracked files and directories. `git clean -di` ----> Open interactive clean mode that will act on directories also.
`git restore`	Discards changes in the working directory or staging area. Can unstage files that have been added to the index (staging area). Restores a file or directory to a previous commit.

	`git restore [options] <path>`
	
	• `--source=<tree>` Specify the source (commit or branch) from which to restore the file.
	• `--staged` Remove changes from the staging area (unstage files).
	• `--worktree` Discard changes in the working directory (default).
	• `--patch` Interactively restore parts of a file.
	• `-s` or `--source` Specify the commit to restore from. Defaults to HEAD.
	
	`git restore --staged css\first.css` ----> Unstage specified file.
	`git restore --staged *` ----> Unstage all files.
	`git restore README.md` ----> Resets README.md to the version in the latest commit (HEAD).
	`git restore --staged --worktree <file>` ----> Discard changes in both the working directory and staging area for a specific file.
	`git restore --source=abc123 README.md` ----> Restores README.md to its state in the commit with hash `abc123`

`git commit`	Commit changes in staging area.
	`git commit -m "A message to be shown for the commission"`
	
	Commit changes directly without staging area (**Should be tracked first**):
	`git commit -a -m "A message to be shown for the commission"`

`git log`	Show commit logs. `git log --oneline` ----> Condenses each commit to a single line. `git log --decorate` ----> Displays the references (branches, tags, etc) that point to each commit. `git log --graph` ----> Draws an ASCII graph representing the branch structure of the commit history. `git log -3` ----> Displays only the 3 most recent commits. `git log --after="2014-7-1" --before="2014-7-4"` ----> Displays commits in specific period. `git log --author="John"` `----> This displays all commits whose author includes the name John.`
`git shortlog`	`It groups each commit by author and displays the first line of each commit message.`
`git reflog`	View the history of changes made to the references in a Git repository, such as HEAD, branches, or tags. It tracks all movements of HEAD (the current commit pointer), including commits, resets, checkouts, merges, and rebases, even if those actions aren't reflected in the commit history. Useful for recovering commits that are no longer part of a branch (e.g., after a `reset --hard`).

Feature	git reflog	git log
Scope	Tracks changes to HEAD and references.	Displays commits in the commit history.
Includes Orphaned Commits	Yes	No
Use Case	Debugging, recovery, and undoing operations.	Viewing commit history and authorship.

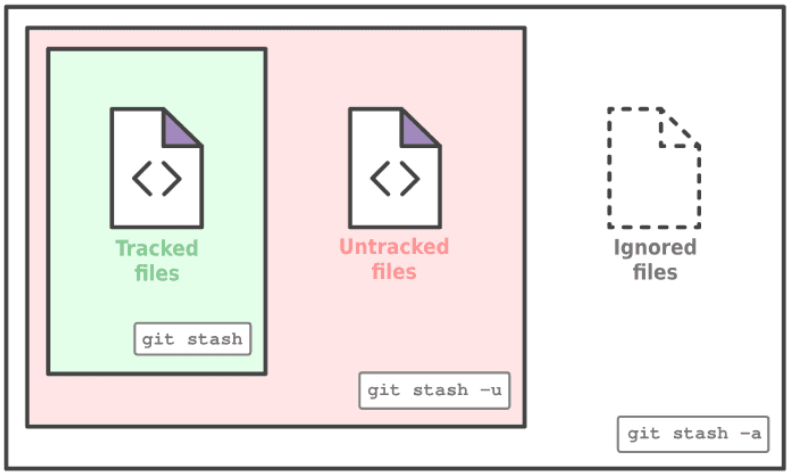
`git reset`	Undo changes by resetting the state of the current branch and optionally the staging area and/or working directory. It can be used to unstage files, undo commits or reset the branch's history to a previous state. `git reset [<mode>] [<commit>]` **Modes of Operation** • `--soft` Moves the branch pointer (HEAD) to the specified commit but keeps changes in the staging area and working directory. • `--mixed` (default) Moves the branch pointer to the specified commit and unstages files while keeping changes in the working directory. • `--hard` Moves the branch pointer to the specified commit and removes changes from the

	staging area and working directory. • `--merge` Similar to `--hard` but keeps uncommitted changes safe if they do not overlap with the reset commit. • `--keep` Resets the branch pointer but keeps uncommitted changes that are not conflicting. `git reset <file>` ----> Unstage files that were added to the staging area `git reset --soft HEAD~1` ----> Undo the last commit but keep changes in the staging area `git reset --mixed HEAD~1` ----> Undo the last commit and unstage the changes `git reset --hard HEAD~1` ----> Undo the last commit and remove changes from both the staging area and working directory `git reset --hard <commit-hash>` ----> Reset the branch to a specific commit

`git revert`	Create a new commit that undoes the changes of a specified commit.
	
	Unlike `git reset`, which removes commits from the history, `git revert` preserves the history by adding a new commit that negates the changes.
	
	`git revert <commit-hash>`
	`git revert <commit1> <commit2>`
	
	If you made a mistake while reverting, you can revert the revert:
	`git revert <revert-commit-hash>`
	
	If there are conflicts during the revert process, resolve them and then complete the revert with:
	`git revert --continue`

`git branch`	Manage branches in a Git repository. `git branch` ----> List all current branches in local repo. `git branch -a` ----> List all local and remote branches. `git branch -r` ----> List remote branches only. `git branch newBranch` ----> Create a branch. `git branch -d b_Name` ----> Delete a branch safely (check for unmerged changes first) `git branch -D branchName` ----> Force delete a branch. `git branch --track <branch-name> <remote>/<branch-name>` ----> Create a local branch tracking a remote branch. `git branch -m newName` ----> Rename opened branch. `git branch --merged` ----> List all branches that have been merged into the current branch. `git branch -v` ----> Displays the last commit for each branch.
`git switch`	Switch between branches. Create new branches and optionally switch to them. Set up tracking for remote branches. `git switch <branch-name>` ----> Switch to existing branch. `git switch -c <new-branch>` ----> Create a new branch from current HEAD and switch to it. `git switch -` ----> Switch Back to the Previous Branch. `git switch --detach <commit-hash>` ----> Switch to a specific commit in a detached HEAD state. `git switch --track origin/feature-1` ----> Creates a local branch `feature-1` that tracks the remote branch `origin/feature-1`
`git checkout`	Switch between branches. Restore files to a previous state from a specific commit or branch. Create new branches (optionally switching to them). This command is gradually being replaced by more specific commands like `git switch` (for branch management) and `git restore` (for file restoration), but `git checkout` is still widely used and supported.

	`git checkout newBranch` ----> Switch to existing branch. `git checkout -b newBranch` ----> Create a new branch from current HEAD and switch to it. `git checkout -b newBr Br1` ----> Create a new branch from Br1 and switch to it. `git checkout <commit-hash>` ----> Moves HEAD to a specific commit, placing you in "detached HEAD" mode, where no branch is checked out.
`git merge`	Combine the changes from one branch into another. `git merge <branch_name>` ----> Combines the `<branch_name>` into the current branch. **Options:** • `--no-ff` Forces a merge commit, even if a fast-forward merge is possible. • `--ff-only` Ensures the merge is fast-forward; fails if not possible. • `--abort` Cancels the merge and reverts to the pre-merge state. • `--squash` Combines all commits from the source branch into a single commit. • `--continue` Completes a merge after resolving conflicts.
`git rebase`	Reapply commits from one branch on top of another, effectively rewriting commit history. `git rebase <base_branch>` ----> Moves the commits of the current branch to the tip of `<base_branch>.` **Options:** • `-i` Interactive rebase for editing commits. • `--continue` Continue rebasing after resolving conflicts. • `--abort` Abort the rebase process and return to the original branch state. • `--skip` Skip the current conflicting commit and continue rebasing.
`git clone`	Create a copy of a remote repository on your local machine. `git clone <repo> <local_directory>` `git clone --branch <tag> <repo>` ----> Clone a specific tag. `git clone -b <branch_name> <repository_url>` ----> Clone a specific branch `git clone -depth=3 <repo>` ----> Only clone the commits specified by depth (Shallow clone).
`git remote`	An interface for managing a list of remote entries that are stored in the repository's ./.git/config file. `git remote` ----> Lists all configured remote repositories. `git remote -v` ----> Include the URL of each connection. `git remote add <name> <url>` ----> Create a new connection to a remote repository. `git remote rm <name>` ----> Remove a configured remote repository. `git remote set-url <name> <new_url>` ----> Updates the URL of an existing remote. `git remote rename <old-name> <new-name>` ----> Renames an existing remote repository.
`git push`	It is used to upload local repository content to a remote repository. `git push <remote> <branch>` ----> Push the specified branch to remote. `git push <remote> --force` ----> Force the push even if it results in a non-fast-forward merge. `git push <remote> --tags` ----> Push All Tags. `git push origin <tag-name>` ----> Push a Specific Tag. `git push origin --delete <branch-name>` ----> Delete a branch from remote repo. `git push origin --delete <tag-name>` ----> Delete a Remote Tag.
`git fetch`	Download the latest changes (commits, files, and branches) from a remote repository but does not automatically merge them into your working branch. `git fetch <remote>` ----> Fetch all of the branches from the repository. `git fetch <remote> <branch>` ----> Only fetch the specified branch. `git fetch --tags` ----> Fetch all tags from the remote. `git fetch --all` ----> Fetch from all configured remotes.
`git pull`	It is used to pull all changes from a remote repository into the branch you are working on. `git pull <remote>` ----> Fetch the specified remote's copy of the current branch and immediately merge it into the local copy.

	Options: • `--rebase` Use rebase instead of merge when applying changes from the remote branch. • `--ff-only` Prevent merge commits if your local branch cannot be fast-forwarded. • `--no-commit` Fetch and merge but stop before committing.
`git stash`	Temporarily save (or "stash") changes in your working directory and staging area without committing them. `git stash` ----> Stashing any tracked files (staged, unstaged) that are not committed yet. `git stash -u` ----> Includes untracked files. `git stash -a` ----> Includes ignored files. Stash Everything (Tracked, Untracked, and Ignored Files)  `git stash save "message on stashes list"` ----> Stashing with customized message. `git stash list` ----> list all stashes IDs. `git stash pop` ----> Un-stashing latest added files. `git stash pop stash@{2}` ----> Un-stashing stash number 2. `git stash apply` ----> Un-stashing a copy of latest added files. `git stash apply stash@{2}` ----> Un-stashing a copy of stash number 2. `git stash drop` ----> Dropping the latest stash with its content. `git stash drop stash@{2}` ----> Dropping stash number 2 with its content. `git stash show` ----> Show the content of latest stash. `git stash show stash@{2}` ----> Show the content of stash number 2. `git stash branch BranchName stash@{2}` ----> Change to `BranchName` branch and un-stashing stash number 2 in it. `git stash clear` ----> Deleting all stashes with their contents.
`git tag`	Create, list, delete, and manage tags in Git. `git tag [options] <tag_name> [<commit>]` • `<tag_name>`: The name of the tag. • `<commit>`: Optional; the commit to tag. **Defaults** to the latest commit on the current branch. `git tag` ----> Display all tags in the repository. `git tag -l "v1.*"` ----> Filter tags matching a pattern. `git tag v1.0.0` ----> Create a Lightweight Tag. `git tag -a <tag_name> -m "Tag message"` ----> Create an Annotated Tag. `git tag -s <tag_name> -m "Tag message"` ----> Create a Signed Tag. `git tag <tag_name> <commit_hash>` ----> Create a Tag for a Specific Commit. `git show <tag_name>` ----> Display details of an annotated tag. `git push origin <tag_name>` ----> Push a single tag. `git push origin --tags` ----> Push all tags. `git tag -d <tag_name>` ----> Delete a tag locally. `git push origin --delete <tag_name>` ----> Delete a tag from a remote repository.
`git check-ignore`	Identify whether a specific file or directory is being ignored by Git based on .gitignore rules and other ignore mechanisms. `git check-ignore [options] <file>...`

Options:

- `-v`, `--verbose` Show the ignore rule and its origin (e.g., .gitignore or global ignore).
- `-z` Output paths with a null character (for scripting).

```
git check-ignore debug.log                  ----> Checking a Single File.
git check-ignore debug.log temp/data.txt    ----> Checking Multiple Files.
git check-ignore -v <file>                  ----> See the exact ignore rule and the .gitignore file responsible.
```

Example Output:

```
.gitignore:3:*.log     debug.log
```

- `.gitignore` is the file where the rule is defined.
- `3` indicates the line number of the rule in .gitignore.

`*.log` is the ignore pattern applied.

About the Author

Connect with me on Linkedin: mohaned-ahmad

Explore more at: GitHub Repository