1.Define VCS.

A system that records changes to a file or set of files over time so that you can recall specific versions later.

2.History of VCS

Local VCS=>

copy files into another directory .

can be error prone since difficult to track the multiple action on directories.

Centralised VCS=>

Helps in collaborating with developers in other system.

(Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.

Distributed VCS=>

(Git, Mercurial or Darcs). Clients fully mirror the repository including git history no dependency on

single server or device.

3.Majorly used Git commands

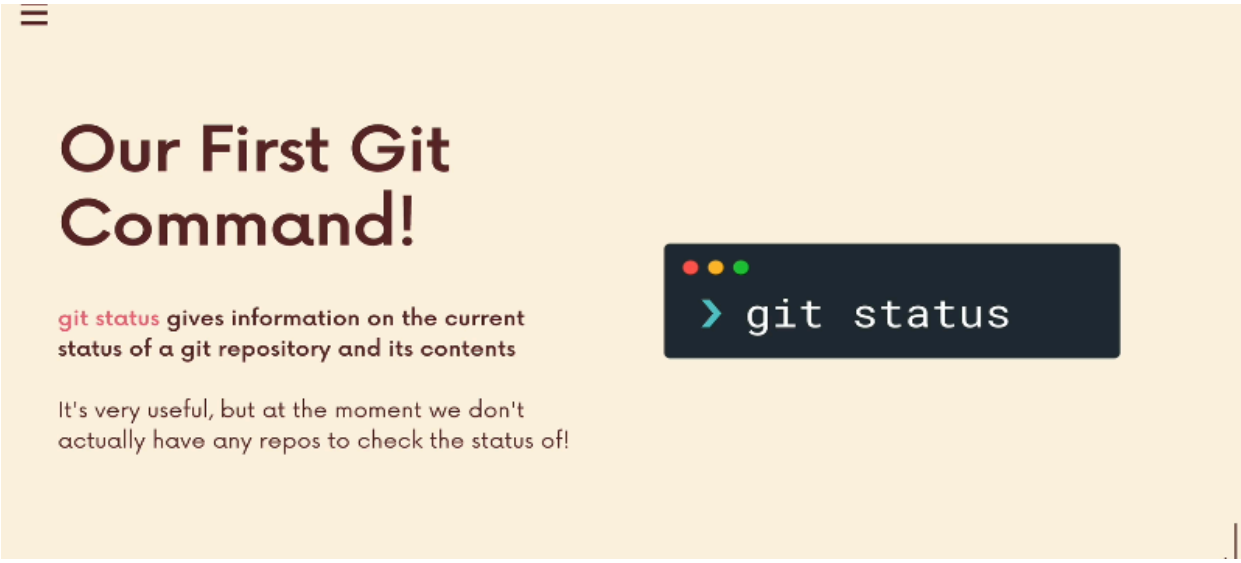
Git init =>

creates git repository inside the directory where the command is executed.

This is hidden .git directory and it contains all the files related to the repository.

Also default branch(mostly main) can be configured while running this comman.

Git status=>



Git clone=>

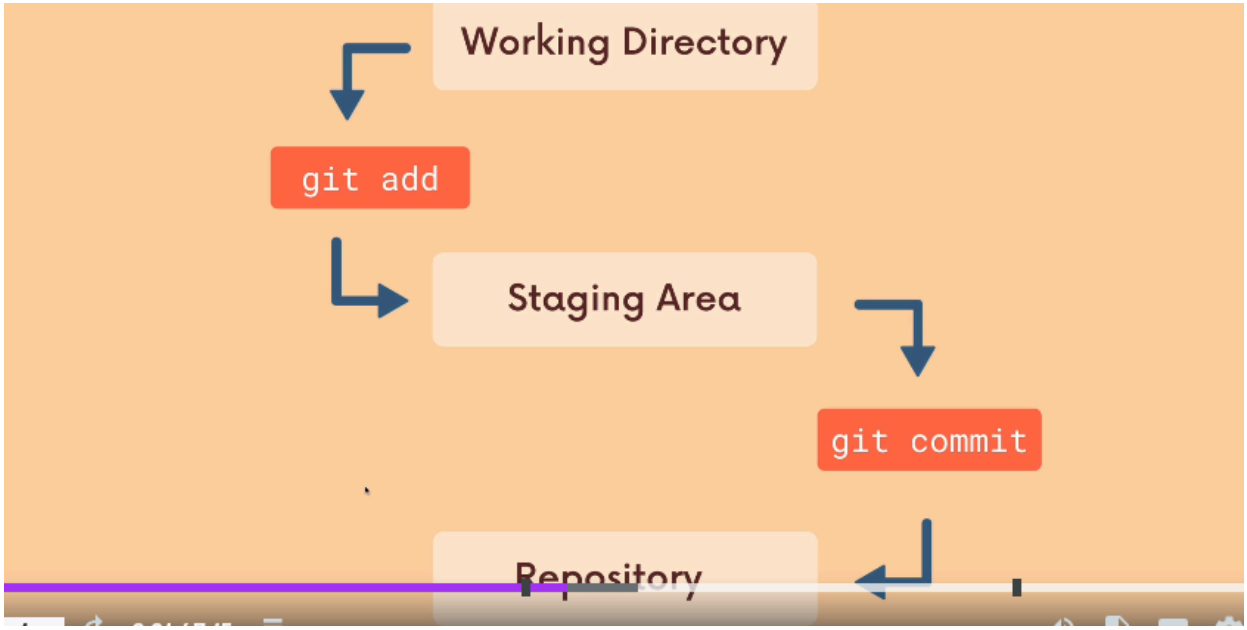
Clones the git repository to the current directory i.e, **git clone <url>**

Git add=>

Adds the files from working directory to the staging directory i.e, **git add .***

Git commit=>

Add the file from staging directory to the remote directory i.e, **git commit -m "msg"**



Amending Commits

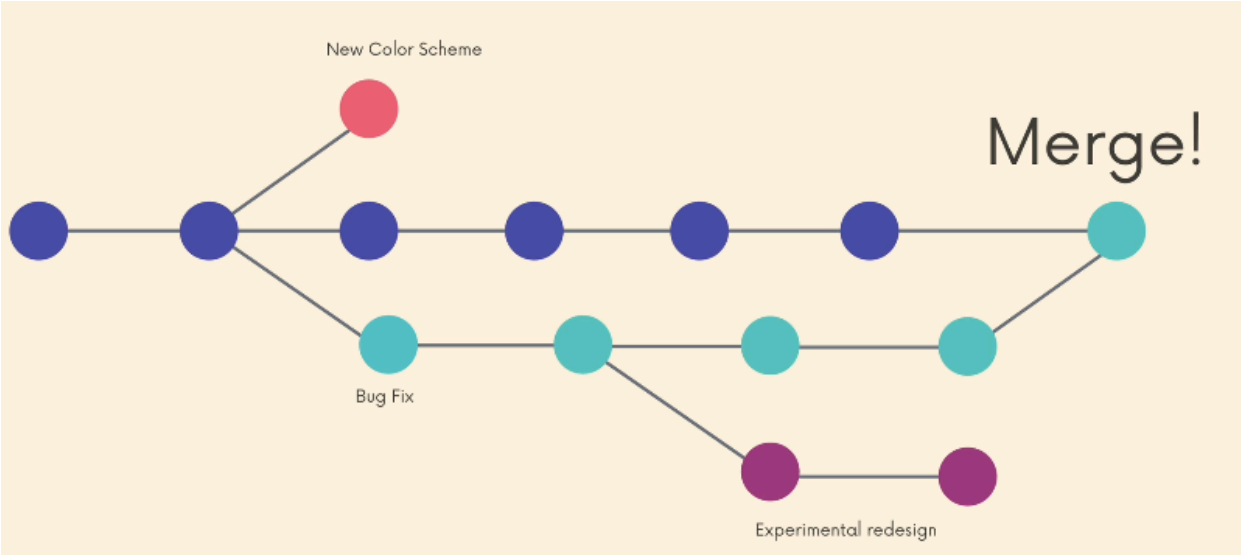
Suppose you just made a commit and then realized you forgot to include a file! Or, maybe you made a typo in the commit message that you want to correct.

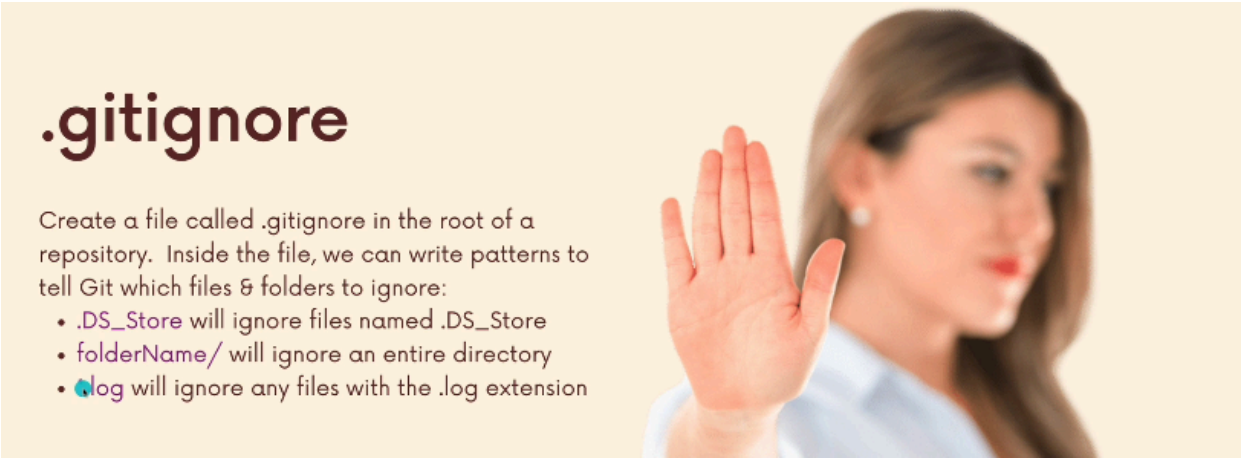
Rather than making a brand new separate commit, you can "redo" the previous commit using the --amend option

```
❯ git commit -m 'some commit'
❯ git add forgotten_file
❯ git commit --amend
```

Git branch=>

Returns List of current branches in our repository. Eg : **git branch**
doesn't return if there is only main branch and no other branch
Creates new branch eg : **git branch <branch-name>**





Git checkout=>

git checkout <branch-name> - switches to the mentioned branch

Git switch branch-name =>

git switch <branch name> - same as git checkout

git switch -c <branch-name>- creates and switch to the branch

Git log => returns the history of changes made in repository.

Git stash=>

Stashing

Use **git stash pop** to remove the most recently stashed changes in your stash and re-apply them to your working copy.

```
❯ git stash pop
```

Git fetch=>

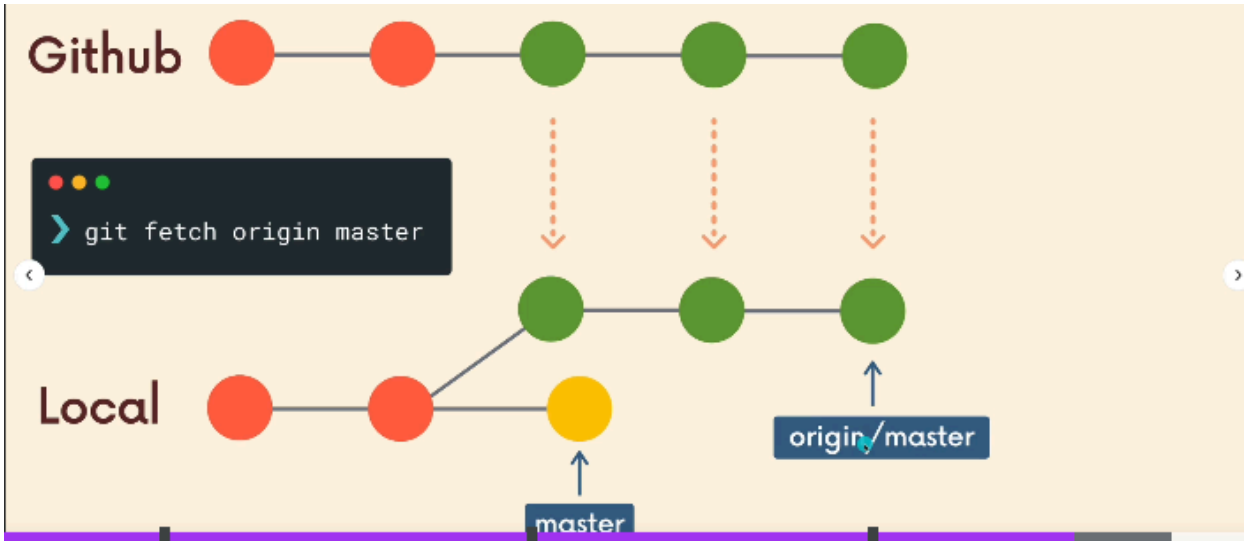
Git Fetch

The **git fetch <remote>** command fetches branches and history from a specific remote repository. It only updates remote tracking branches.

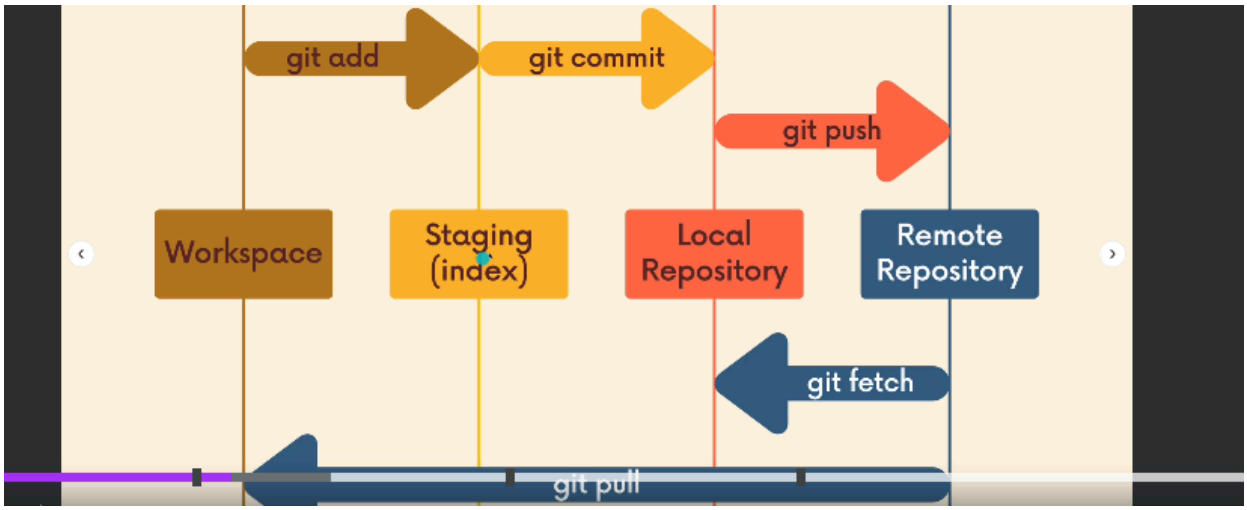
git fetch origin would fetch all changes from the origin remote repository.

```
❯ git fetch <remote>
```

If not specified, <remote> defaults to origin



Git pull=>



Git push=>

Git diff=>

git diff

Without additional options, **git diff** lists all the changes in our working directory that are NOT staged for the next commit.

```
❯ git diff
```

Compares Staging Area and Working Directory

Compared Files

For each comparison, Git explains which files it is comparing. Usually this is two versions of the same file.

Git also declares one file as "A" and the other as "B".

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

Git Merge=>

Merging Made Easy

To merge, follow these basic steps:

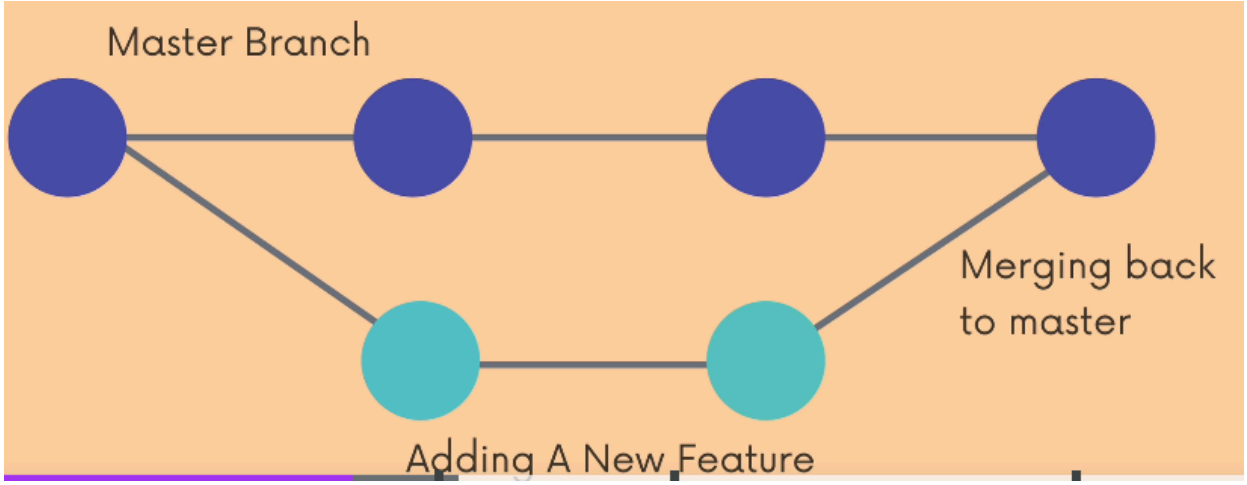
1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)
2. Use the **git merge** command to merge changes from a specific branch into the current branch.

To merge the bugfix branch into master...

```
❯ git switch master
❯ git merge bugfix
```

4.Branch

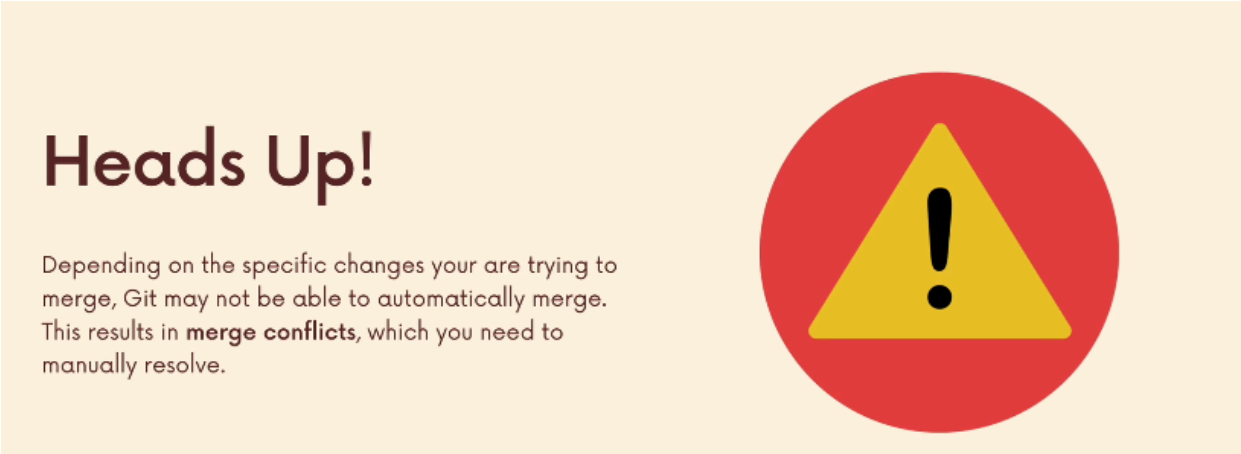
Branching means you diverge from the main line of development and continue to do work without messing with that main line.



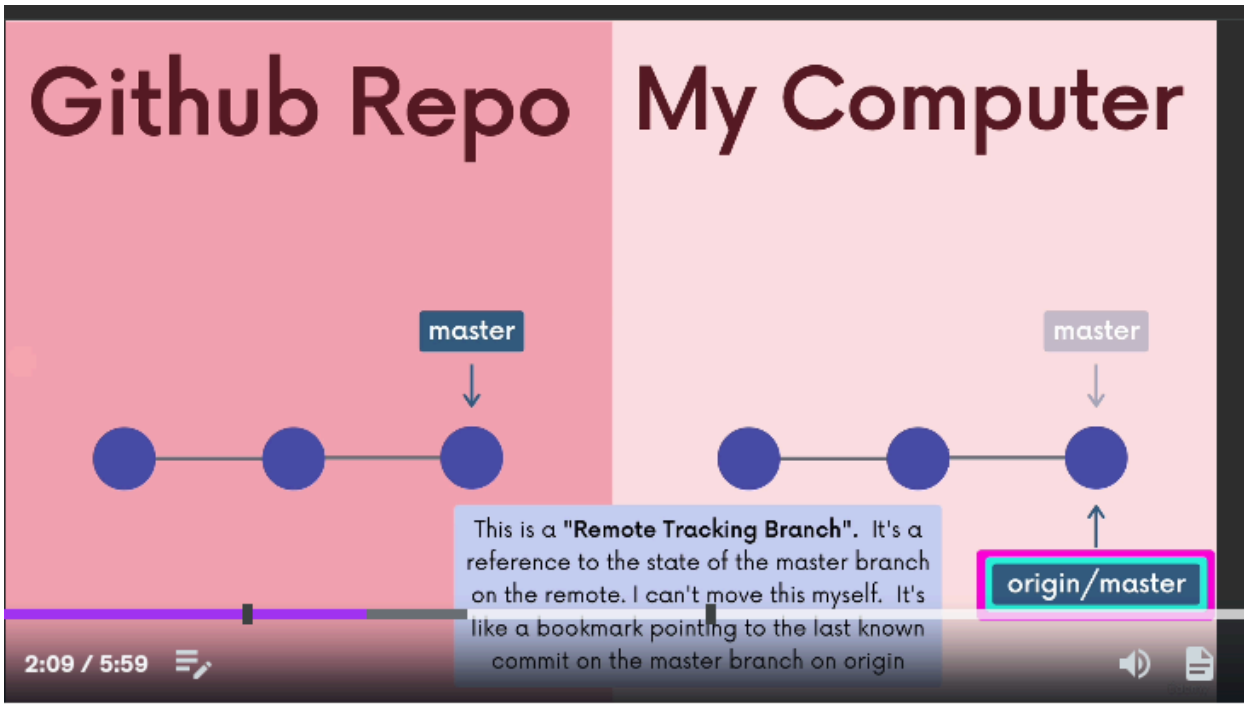
5.HEAD

Pointers refer to a current location in a repository.
Points to a particular branch reference.

6.Merge Conflict

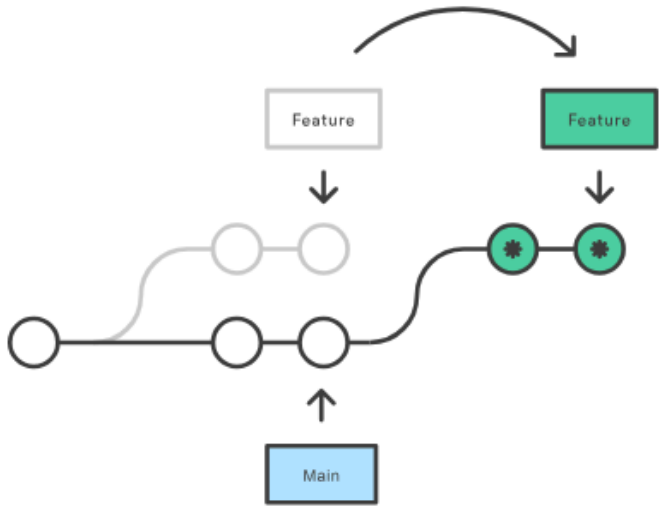


7. Remote Tracking Branches



8.Rebasing
Re-writing commit history

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:

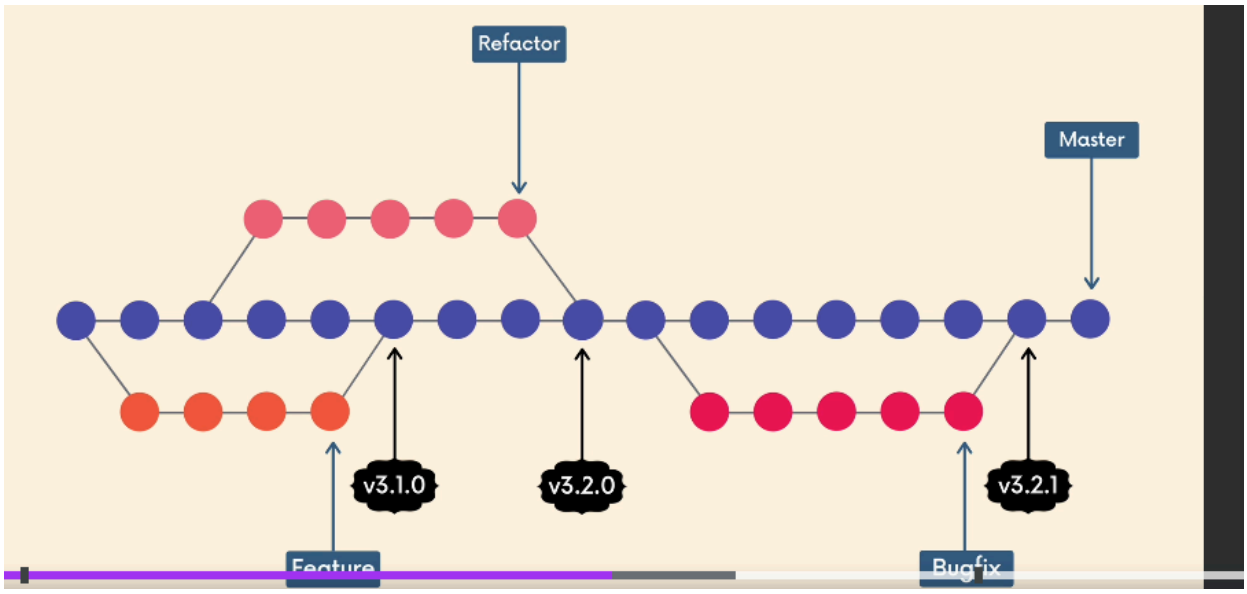


9.Git tags

Git Tags

Tags are pointers that refer to particular points in Git history. We can mark a particular moment in time with a tag. Tags are most often used to mark version releases in projects (v4.1.0, v4.1.1, etc.)

Think of tags as branch references that do NOT CHANGE. Once a tag is created, it always refers to the same commit. It's just a label for a commit.



The Two Types

There are two types of Git tags we can use: lightweight and annotated tags

lightweight tags are...lightweight. They are just a name/label that points to a particular commit.

annotated tags store extra meta data including the author's name and email, the date, and a tagging message (like a commit message)

Viewing Tags

git tag will print a list of all the tags in the current repository.

```
> git tag
```

10.Reflogs

reflogs

Git keeps a record of when the tips of branches and other references were updated in the repo.

We can view and update these reference logs using the **git reflog** command

Git only keeps reflogs on your **local** activity. They are not shared with collaborators.

Reflogs also expire. Git cleans out old entries after around 90 days, though this can be

11.Forking

Fork means to make a copy of the repository (the one being forked) into my own githul
I want to fork the official jQuery repository, then I would go to https://github.com/jquery/
hit the "Fork" button and GitHub will copy the repository (jquery) to my account (http://
sanjaykhadka). Then a copied version of that repository will be available to me at http:/
github.com/sanjaykhadka/jquery

Now I can make whatever the changes I wish to make to my repository and then send
request to the original repository (jQuery's repository), asking the jQuery team to merg

12.Detached head - when HEAD is pointing to commit rather than branches.

Discarding Changes

Suppose you've made some changes to a file but don't want to keep them. To revert the file back to whatever it looked like when you last committed, you can use:

git checkout HEAD <filename> to discard any changes in that file, reverting back to the HEAD.

```
❯ git checkout HEAD <file>
```

Unmodifying Files with Restore

Suppose you've made some changes to a file since your last commit. You've saved the file but then realize you definitely do NOT want those changes anymore!

To restore the file to the contents in the HEAD, use **git restore <file-name>**

```
❯ git restore <file-name>
```

NOTE: The above command is not "undoable" If you have uncommited changes in the file, they will be lost!

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   cat.txt
        modified:   dog.txt
        new file:   secrets.txt

UndoingStuff ❯ git restore --staged secrets.txt                    master
UndoingStuff ❯ git status                                          master
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   cat.txt
        modified:   dog.txt

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


Git Reset

Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commits, you can use **git reset.**

git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone

```
❯ git reset <commit-hash>
```

Git Revert

git revert is similar to **git reset** in that they both "undo" changes, but they accomplish it in different ways.

git reset actually moves the branch pointer backwards, eliminating commits.

git revert instead creates a brand new commit which reverses/undos the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

```
❯ git revert <commit-hash>
```