**What is Git?**

Git is world’s most important version control system. Here version control is a software that tracks and manages the changes in the files over time. (about 95% users over the world use)

GIT – global information tracker(git means the unpleasant person)

*Version Control* generally allows users revisiting, going back, undoing and comparing the changes in the files.

Git is just one of the version control system. Other version control system include Subversion ,CVS and mercurial.

---------------------------------------------------------------------------------------------------------------------

**Git ≠ Github**

🡪Git is a version control system software that runs in your machine locally and that does not require internet connection.

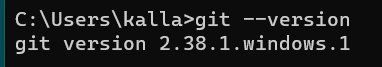
🡪Github is a service/platform that hosts the git repositories to the cloud and makes our works easier.

Git is generally accessed using the command line. But due to the vast popularity of the git the GUI’s of git are also developed. The guis of git are GitHub Desktop ,Source Tree, Tower, GitKraken etc…,

Git is designed to run on a UNIX based interface.

Whenever you install git in your local system,

**You can verify the git version that is installed in your local system using**



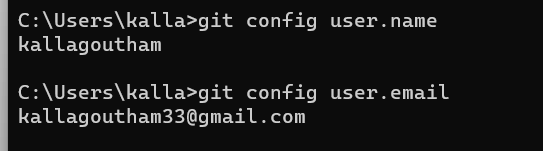
**git -log** 🡪 gives out the log of the git repository

whenever we have installed git in our system we have to configure our github

**git config --global user.name <Your Name>** 🡪 to set the git user.name

**git config --global user.email <your email>**  🡪to set the git email

to verify the config that was set



**start .** 🡪 opens the file explorer from the bash.

**touch <filename>** 🡪command that helps creates the file in the directory.

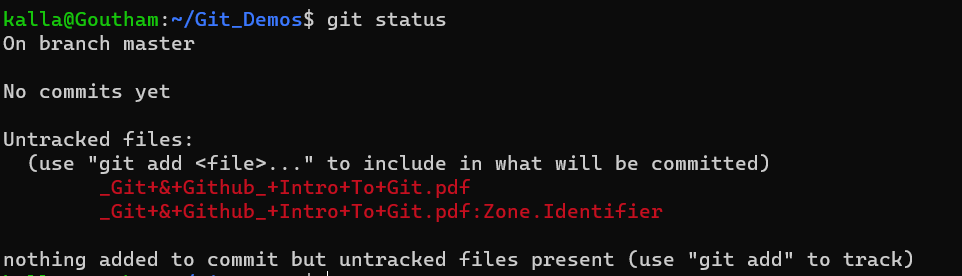
**mkdir <directory/foldername>** 🡪is used to create a directory in the given path.

**rmdir <directory/foldername>** 🡪is used to delete the directory.

**rm <filename>** 🡪removes the file with filename.[-rf are options r🡪recursive and f🡪 force for folders]

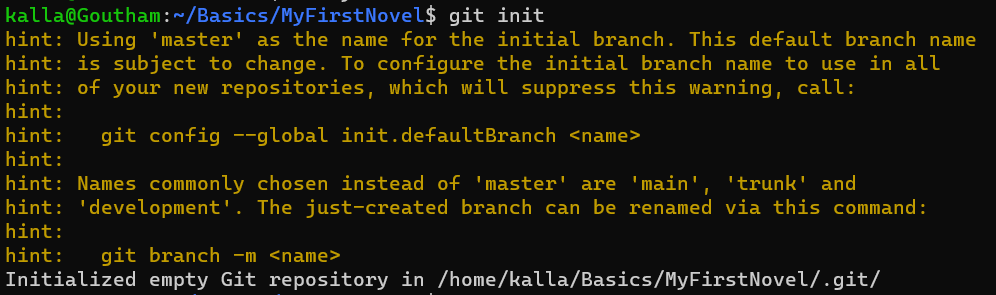
A ***Git Repo*** is a workspace that manages and tracks the files within in folder of the git.

*git status* gives the current status of the git repository and its contents.



*git init* initializes a new git repository wherever we are in the folder.

It creates a .git folder in the folder





Git makes or tracks the changes of the folder and sub folders in where the .git in the folder and all the nested folders in the parent folder are get tracked by the Git.

\*\*Warning: Don’t init a git repo in the git repo as it becomes the git tracking a git which leads to some problems in the future.

The Mysterious .git folder gets created whenever we initialize a git repo.

The commit is an action in the git where I can group some of the work that has done and saved.

we can group some changes and commit.

git add we will select the changes that has to be grouped in the next commited and by doing git add the changes are transferred to an area called staging area whenever before adding the changes are in our working area.

By doing git commit the changes are commited and a checkpoint is created in our git repository and the changes are moved from the staging area to the repository.

git commit

Repository

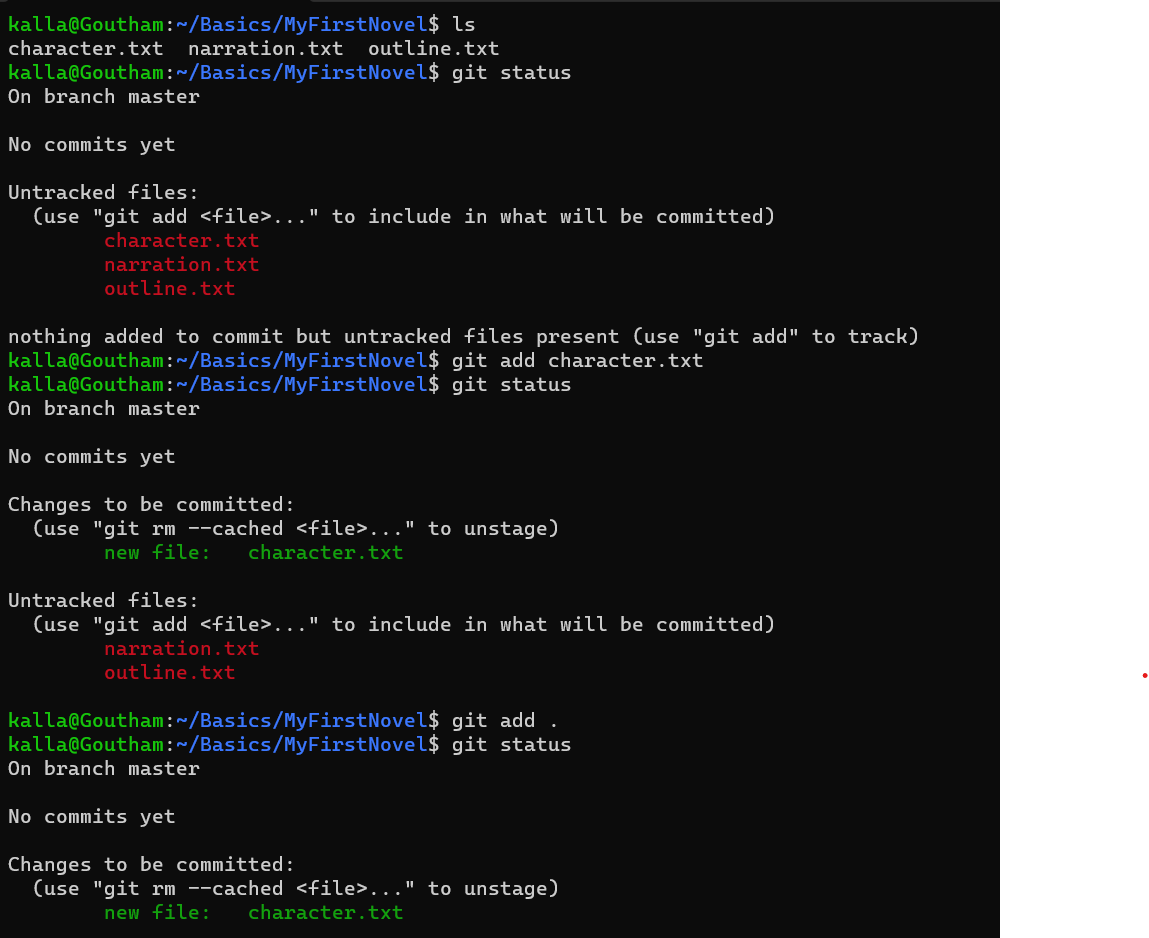
git add

Staging Area

Working Directory

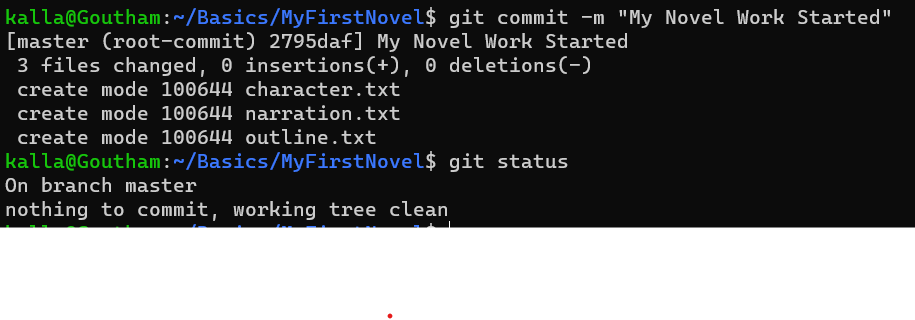
**git add file1 file2**  **🡪** adds the file1 and file 2 to the staging area of the git .

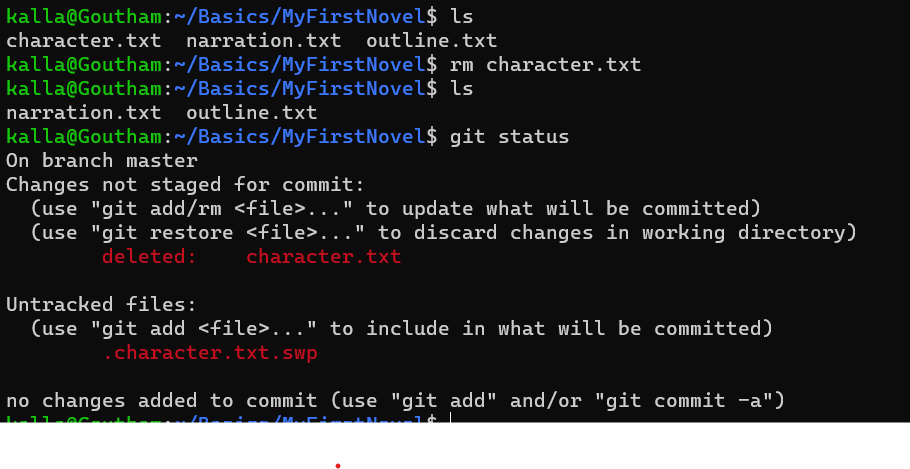
**git add .** **🡪** adds all the untracked files to the git staging area.



we use the git commit in order to actually commit the changes from staging area to the git repository.

**git commit -m “my message”** 🡪is used to commit the changes to the git folder with the given message.

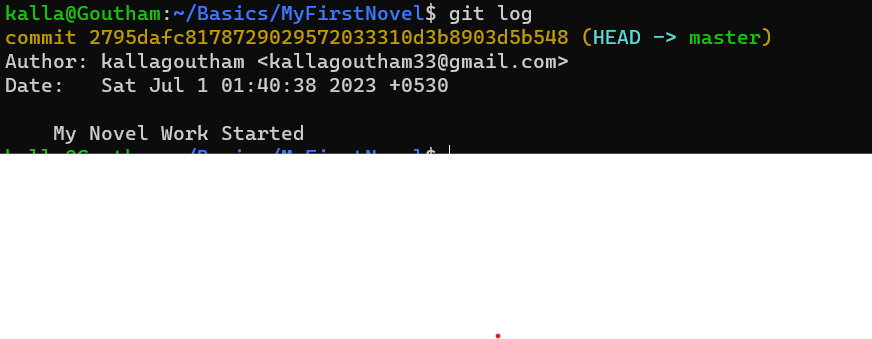


git has two types of files they are untracked files and the modified files if we modify the files in the git folder they are referred as modified files and if they are added then they are called untracked files. 

*git log* is used give the log of the commits that are performed in the git repository.

git log master..branch-X 🡪That will show you commits that branch-X has but master doesn't.

git log master...branch-X If you want to find commit that are either in master or branch-X but not both.

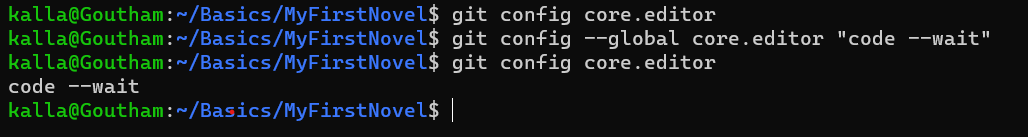


The best practice is to keep commits in the git should be atomic (atomic refers the commit must focus on a single thing or a feature)

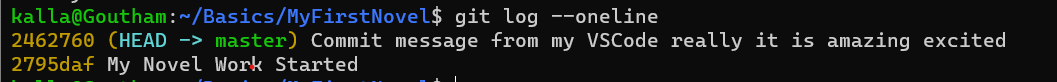
The Git official documentation suggests that the commit messages must be in present tense in the imperative mood. Like make xyz to abc.

The git commit by default opens the vim editor to enter the commit message but we can configure it with the editor of our choice.

We can do that by setting the config of our git.

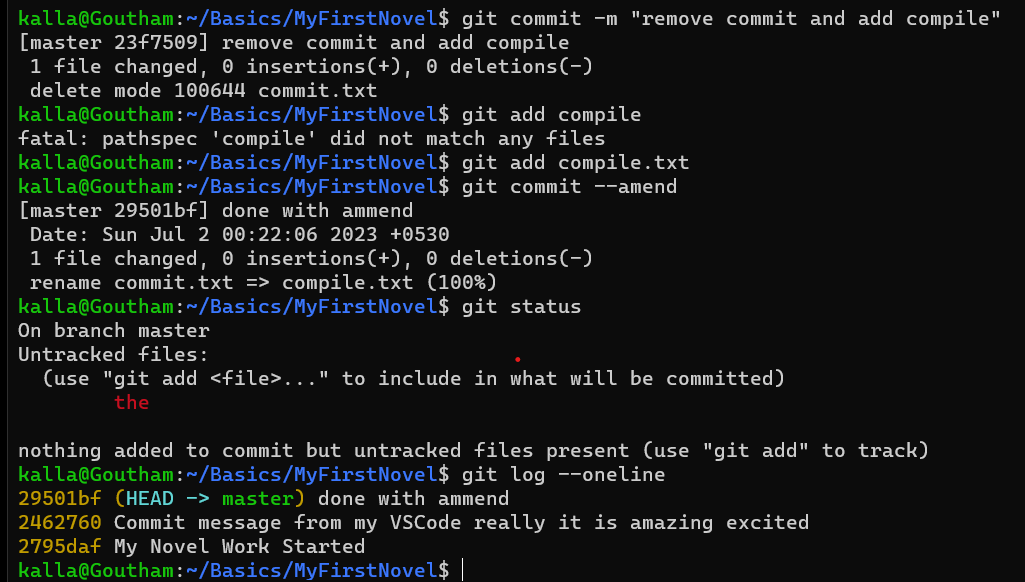


We may end up writing a long para of commit messages in the git we can also tweek only a single line of commit message or to be hidden to be displayed but the actual commit message stays the same.



Stage is also known as the add in the git.

Let us suppose you have changed the working directory and done some work and want to commit them then you go and add and commit them to your git then suddenly you reminded that you have forgotten to add a file or there was a typo in the commit message now Rather than doing a separate new commit you can redo the previous commit using the ***--amend*** option.



We can also tell the git which files and directories to ignore in a given repository using .gitignore file. This is useful for files you know you NEVER want to commit like Secrets, API keys,

Credentials, operating system files, log files, dependencies and packages.

Every time we make a commit in the git we get generated with the weird looking hash with parent and a message. The algorithm used is SHA1 for hash generation and there must be a parent commit for each commit.

---------------------------------------------------------------------------------------------------------------------

***Branches in Git***

Branches in the git are also referred as two different timelines in the git repo/ project. They enable us to work on the two separate contexts where we can try new things or even work on multiple things and features in parallel.

\*\*Disclaimer: The changes I do in one of the branch does not affect another branch.

Branches are independent and they have separate contexts and we can merge them in order to club the ideas or features. The default branch in the git is master branch.

*\*\*\*HISTORY: In 2020 the github changed the default working branch from master to main but the git kept the master as only the default branch.*

---------------------------------------------------------------------------------------------------------------------

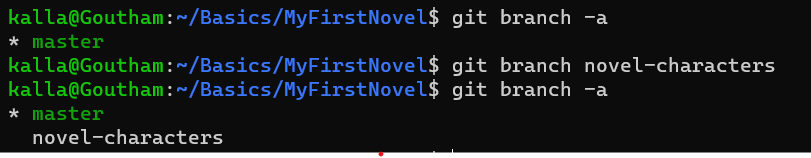
HEAD 🡪master

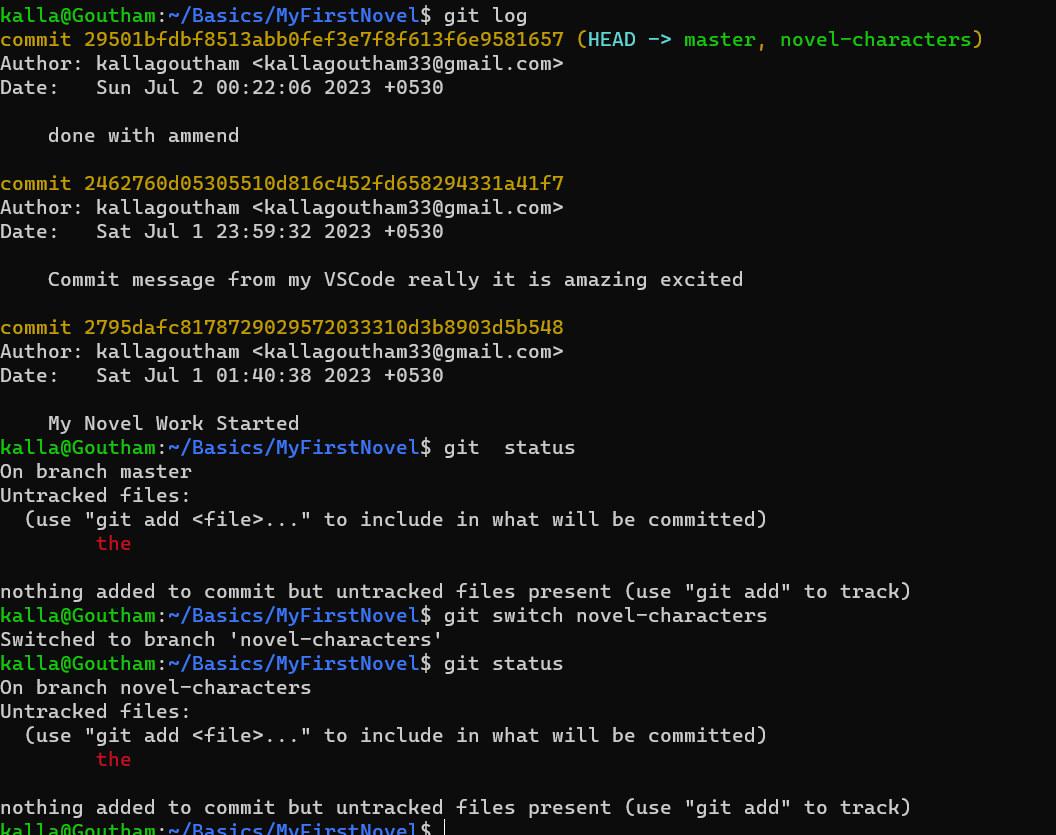
In git the term HEAD refers to a pointer of the current location in your repository. It points to a particular branch reference. In general the HEAD points to the latest commit in any branch if we merge a branch to another branch the last commit in the branch will be head those are also known as tips (edges /latest commits are also referred as HEAD’s) and we navigate or move using that HEAD.

We use git branch -a to list all the branches in the git repo.

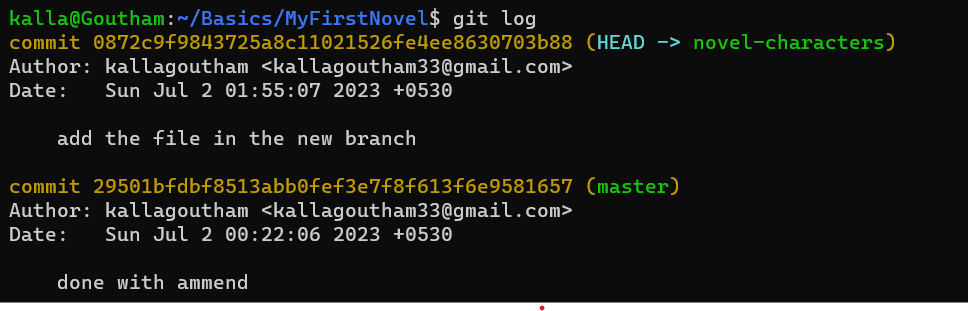


git branch <branch-name> with no spaces in the branch-name creates a branch in the git repo.



git switch <branch-name> is used to switch to a branch with branch-name in the git repo. 

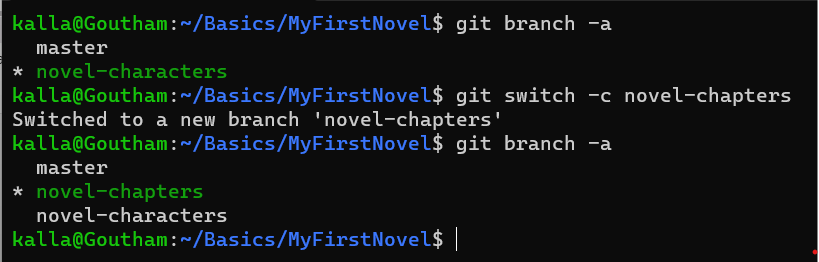
now an example where a latest commit is made to a new branch to a git and the HEAD is pointing to the latest commit on that branch.



git commit -a -m “message” ---- > adds all the staged files and commits just a shortcut.

git checkout <branch-name> ----- >used to switch the branch or restore working tree files or creates and switches.

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

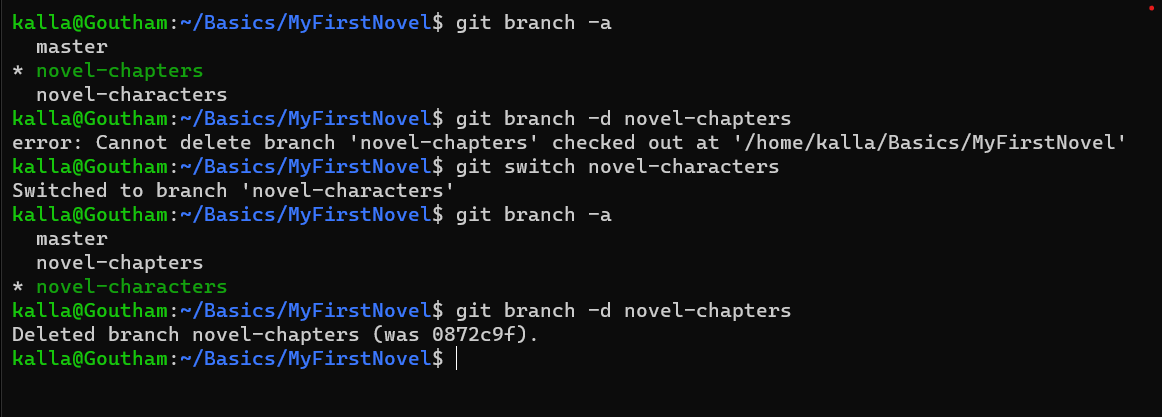


We cant switch between the branches if there was untracked changes we have to commit or stash the changes before we are switching the branches.

**Deleting/Renaming Branches in Git:**

If we want to delete the git branches two conditions must be achieved:

* We should not be on the branch on which we want to delete.
* We should merge all the changes to the upstream before deleting or we can do a force delete.

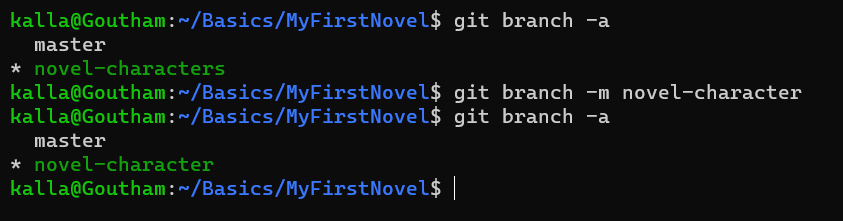


git branch -d <branch-name> 🡪deletes the branch with branch-name.

git branch -D <branch-name> 🡪force deletes branch with branch-name.

to rename a branch we have to be on the branch on which we want to rename

git branch -m <branch-name> 🡪changes the name of the branch on which we are to branch-name.



**Merging Branches in Git:**

Whenever we have different works in different branches and we want those two works/features to be used then we should merge the branches inorder to get both of those works.

MERGING OF BRANCHES CAN BE DONE ON

* Only 2 branches but cannot be performed on two specific commits.
* We always merge the current HEAD branch.

Whenever we want to merge two branches we have to switch to the branch where we want to merge and we have to click git merge <branch-name>.

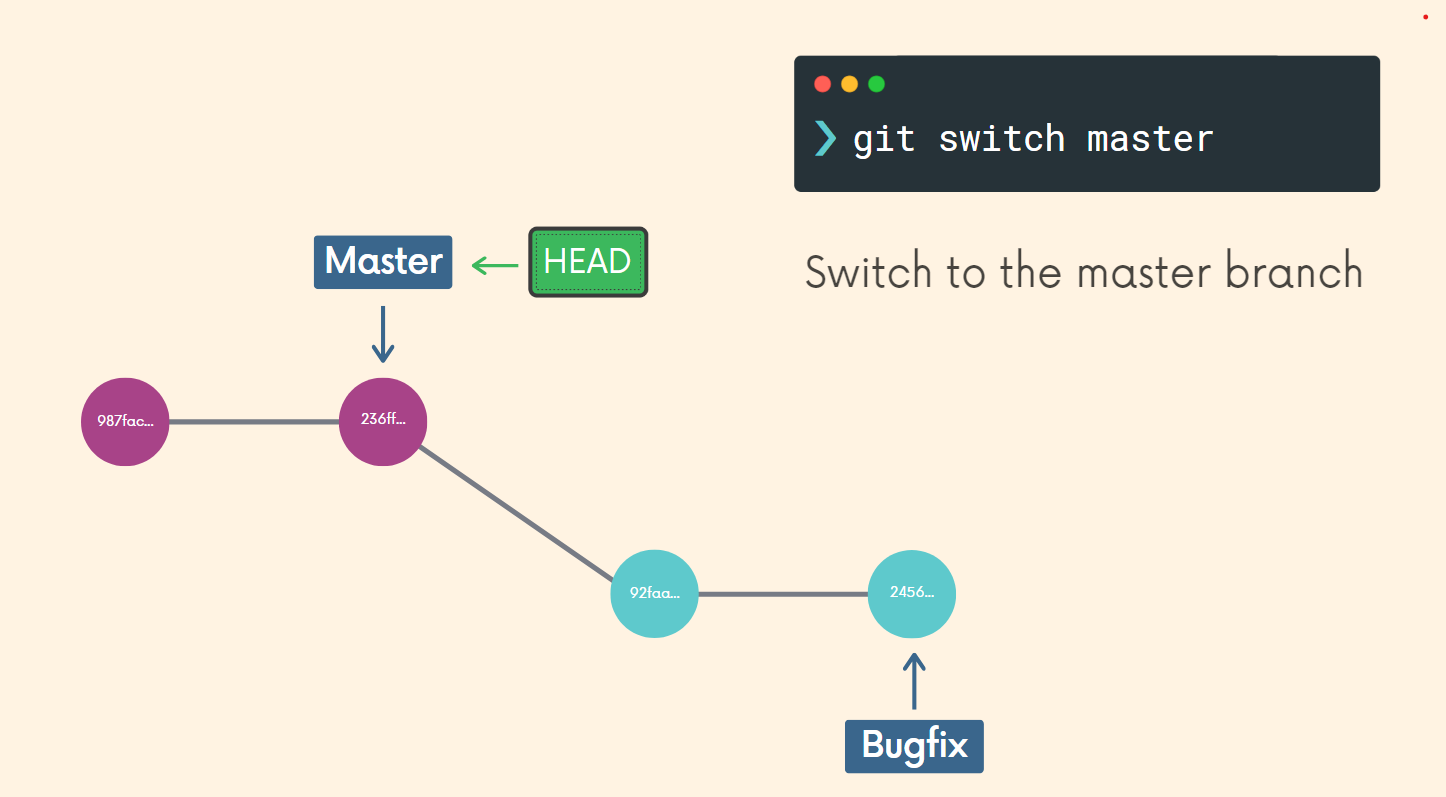
Ex. git switch master

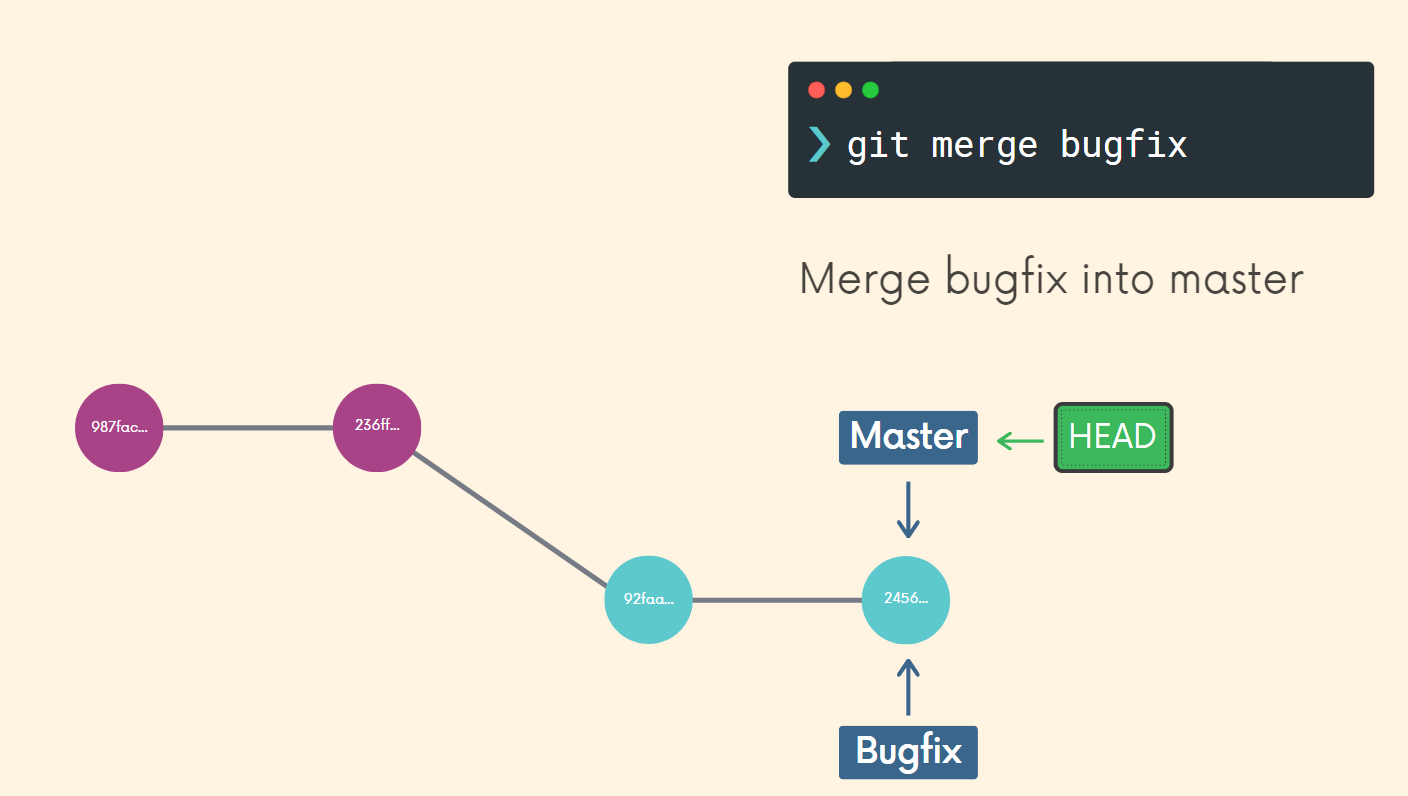
git merge bug-fixes

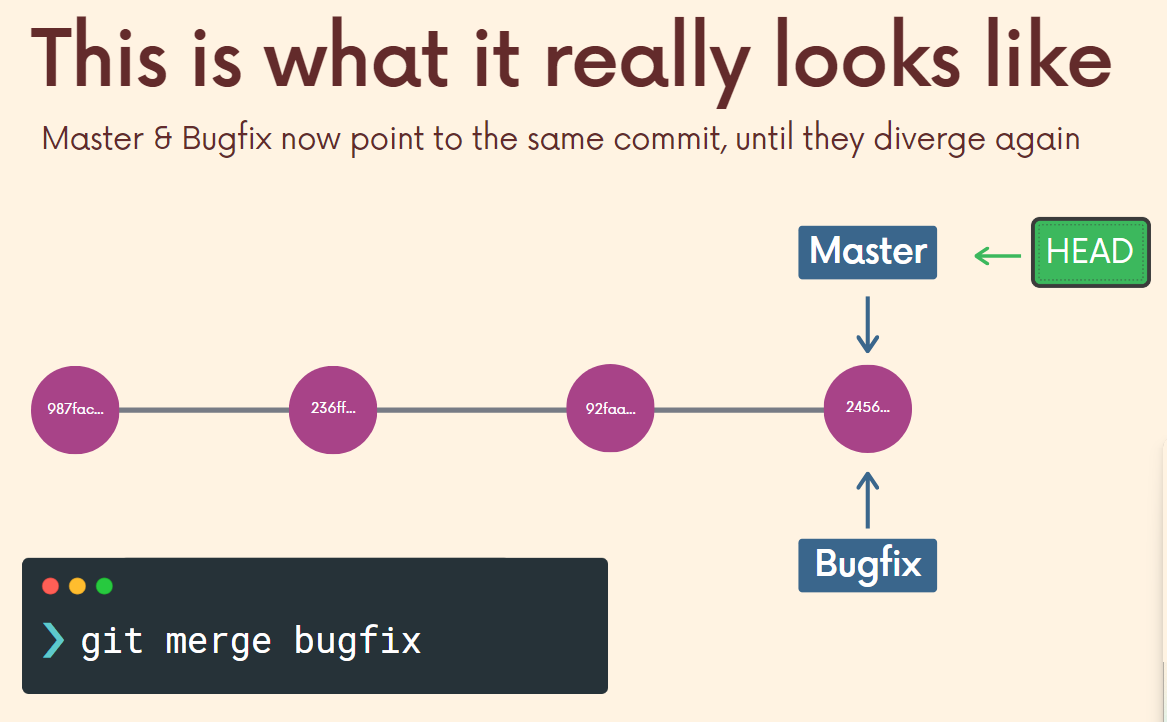
The above commands switches to master branch and merges the bug-fixes branch to the master branch.

There are different types of merging of branches.

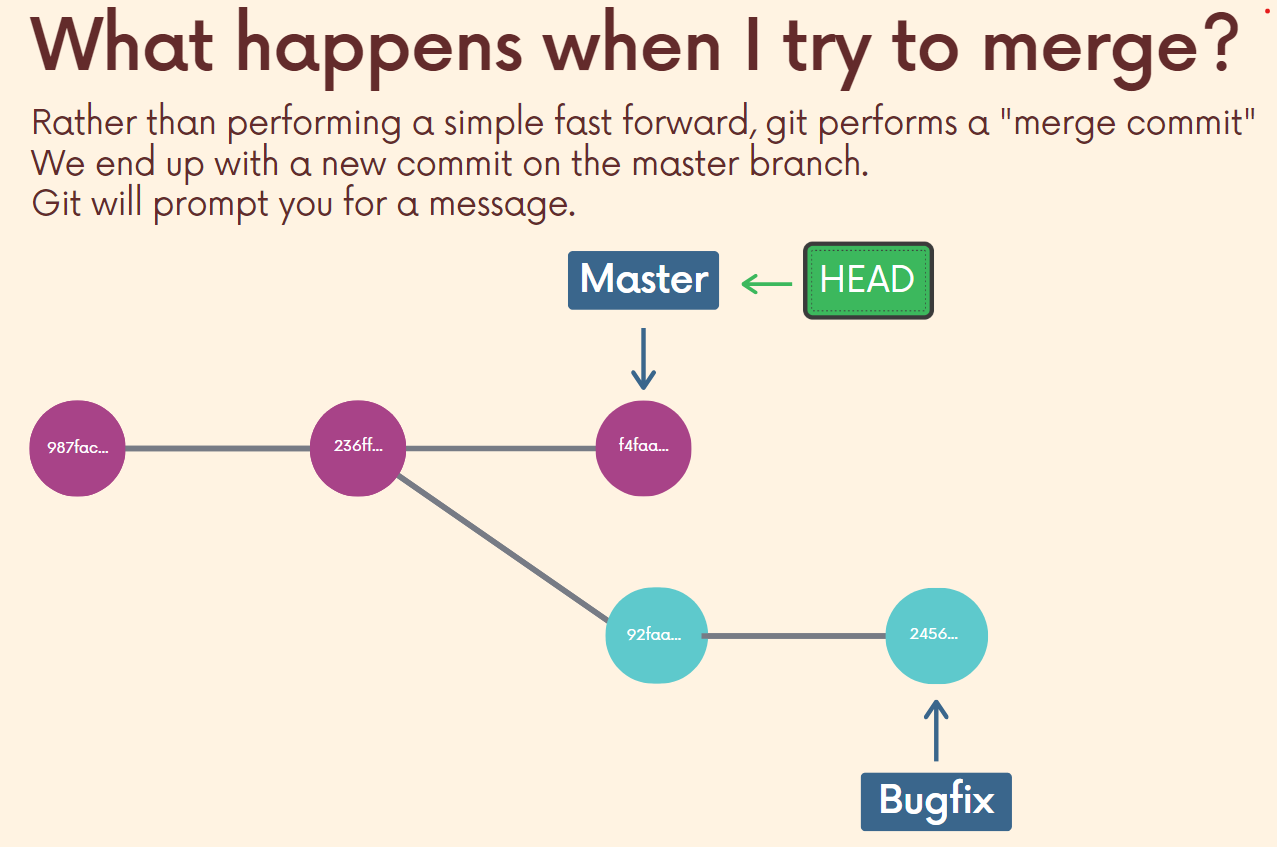
***1.Fast Forwarding Merging:*** In this type of merging the branch that is going to merge is some commits ahead of the branch and on merging with no conflicts its get merged and the HEAD is shifted to the latest commit.

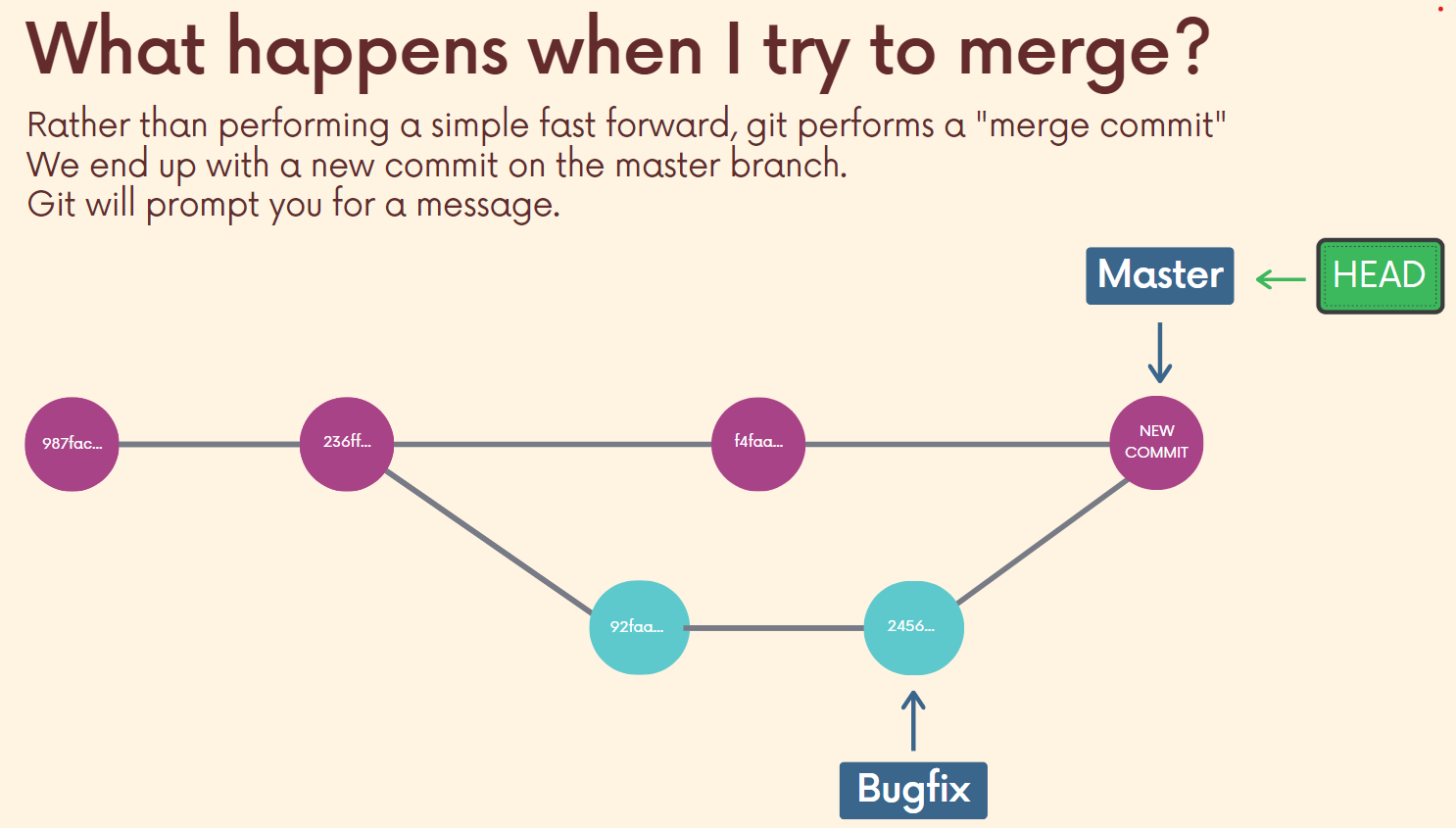






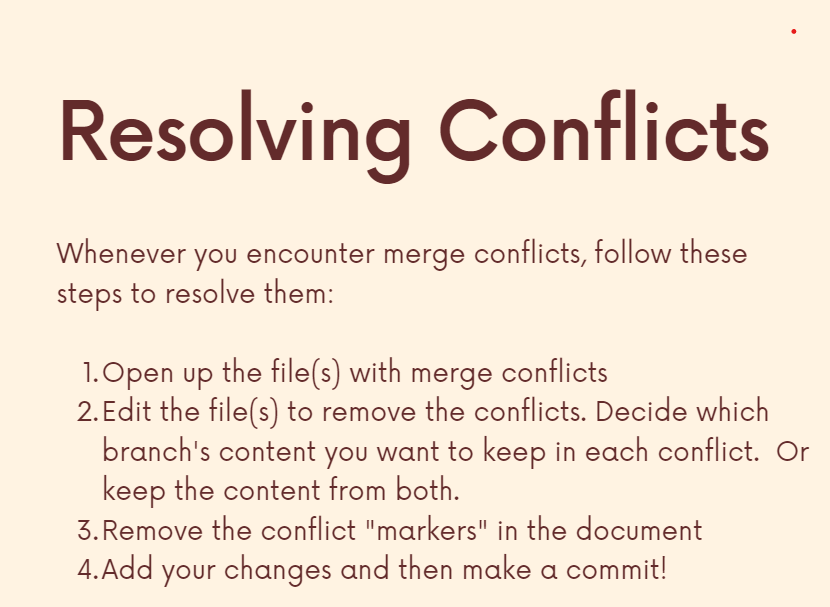
***2.Recursive Strategy Merge:*** It is type of merge where the master is branch or destination branch is some commits ahead and the other branch is not aware of that changes and we don’t have any merge conflicts in that scenario that is referred as Recursive Strategy Merge. In this type of merge since there is no merge conflict it will succeed and branches will autmerge and create a new commit.





***3.Merge Conflicts:*** When git cannot perform auto merging where the changes refer to the same file or the same line then we have to resolve the merge conflicts and we have to add and merge branches. Whenever merge conflicts arises conflict markers arise in the file. This merge conflict markers contains the content from the HEAD branch and the content from the branch which is going to be merged. Then we have to open the files and resolve the merge conflicts and make sure to commit them again.





**Git Diff:**

Git diff is a command used to view the changes between the commits, files, our working directory and more.

**git diff** is same as **git diff HEAD**

git diff shows the changes between the working directory and unstaged changed changes but git diff HEAD shows the difference between the working directory and the HEAD after commiting git diff will give nothing as the changes move to the staging area.

git diff --staged 🡪 gives the changes between the both staged area and the current working directory.

git diff --staged <filename> 🡪 gives the changes between the staged area and the current working directory in the particular file named filename.

git diff <branch1>..<branch2> 🡪 gives the changes /compares the tips of the branches branch1 and branch2 of a repo.

git diff <commit1>..<commit2> 🡪 compares the two different commits of the branch here we have to give the hash values of the commits.

git diff <commit1>...<commit2> 🡪to find the diff from their common ancestor to test, you can use three dots instead of two

git diff HEAD HEAD~1 🡪 gives the diff of the HEAD with the previous commit of HEAD.

***Git Stash:***

Let us suppose that we are working on one of the branch and we want to switch to another branch. Now there are two scenarios in this case either:

* The changes come along with us to the destination branch
* git wont allow us to switch the branches due to potential conflicts if we over write the existing files.

Git stash means it provides a way to stashing these uncommitted changes so that later we can come back to them without giving the unnecessary commits.

git stash 🡪 it helps you save the changes that are uncommitted and the changes are also reverted in your working directory and you can come back to them later.

git stash is same as git stash save

git stash pop 🡪it helps them to get all the stashed changes to my current directory here the changes are removed are removed from stash.

git stash apply 🡪it helps them to apply all the stashed changes to my current directory here the changes will stay in the stash and we can apply and use them in multiple branches.

git stash list 🡪 lists all the stashes made.

git stash apply stash@{2} 🡪 In this we can apply a particular reference of the stash.

git stash drop stash@{2} 🡪removes the stash any how if git stash pop removes but if we want to remove manually we have to use this.

git stash clear 🡪 used to clear all the stash stack.

Note\*: git stash uses the stack data structure so we can make as many as stashes and this STACK data structure uses the LIFO(Last In First Out). The git stash stack is pushed into the remote repo.

***Git Checkout***

Git checkout not only switches the branch with git checkout we can also checkout to the particular commit in the repo.

If we checkout to the back commit then the HEAD will be in detached state.

Detaching means the HEAD refers to a tip of the branch in general but when it points to the commit other than the tip then it is known as the detached HEAD.

git checkout commit <commit-hash> 🡪 switches or travels back to the particular hash it detaches the HEAD.

git checkout <branch-name> 🡪 switches to the branch.

So to make the detached HEAD to normal we can just use git switch <branch-name> to get to the normal position.

Git checkout to a particular time / commit helps us in creating a branch from that point so that it is easy and makes the collaboration and adding the features easy.

git checkout HEAD~n 🡪 switches or travels back to the commit that is n commits behind the HEAD. This is another way of checkout from HEAD. We can use commit hash or the this ~n to goto the particular commit.

Git checkout is also used to DISCARDING CHANGES in the current working directory. Whenever we made some 1000 files and 1000 lines of changes then manually deleting all the files is not so good idea then we can use:

git checkout HEAD <filename(s)> 🡪 reverts all the changes and takes us back.

git checkout -- <filename(s)> 🡪 does the same actions like above command do.

***Git Restore:***

Git Restore can be used for many purposes. They are :

1. git restore will help us in undoing the changes / un-modifying the changes made to the current working directory.

git restore <filename(s)> 🡪 discards the changes but the thing is that the changes will be lost and you cant get them back.

Git restore generally uses the HEAD as the source and makes the changes the according to that.

But we can also specify the source in the git restore.

git restore --source HEAD~1 <filename(s)> 🡪 restores the files according to the commit which is one commit behind the HEAD and making it as the source in the files specified.

git restore <filename(s)> 🡪 restores the file to the latest commit.

1. git restore is also used to unstage the changes made to the current working directory inorder so that they wont get reflected in the next commit.

git restore --staged <filename(s)> 🡪 removes the file(s) from the staged area.

***Git Reset:***

Git reset command in git is used to reset the repo back to the particular commit.

git reset <commit-hash> 🡪 will reset the repository to the commit-hash specified.

git reset will only removes the commit and resets the repo to the commit-hash but the changes are also carried with us when we move back in the repo if we want to get rid of those changes then we can make a new branch and commit them there and come back or else we can also do

git reset --hard so that the changes will not be carried with us.

git reset --hard <commit-hash> 🡪 resets the repository to the particular hash and the changes are not brought with us.

Instead of commit-hash we can also use the HEAD~n to go n commits back from the reference part.

***Git Revert:***

Git Revert does the same thing as the git reset, but thing is git reset does just moves the pointer to the commit specified whereas the git revert makes a brand new commit and moves the pointer to the brand new commit. In git reset if we switch the branch again the HEAD will point to the tip but in git revert it will point to the changes that what we have done.

git revert <commit-hash> 🡪undo’s the changes and the creates a brand new commit.

git revert HEAD~3 **🡪**Revert the changes specified by the fourth last commit in HEAD and create a new commit with the reverted changes.

git revert -n master~5..master~2 **🡪**Revert the changes done by commits from the fifth last commit in master (included) to the third last commit in master (included), but do not create any commit with the reverted changes. The revert only modifies the working tree and the index.

***Git Clone:***

Git clone is used to make a local copy of the hosted repo in any of the code hosting services like GitHub, GitLab, BitBucket etc..,

git clone <url> 🡪 clones and makes a local copy of the hosted repository or project that is present. The git clone is not GitHub specific but that is git specific. We can clone public repos.

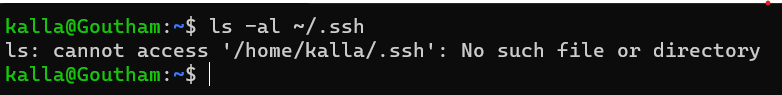
GitHub helps in Collaboration, Exposure, Make Friends, Free Hosting of the projects.

SSH key is also known as SecureShell Key.

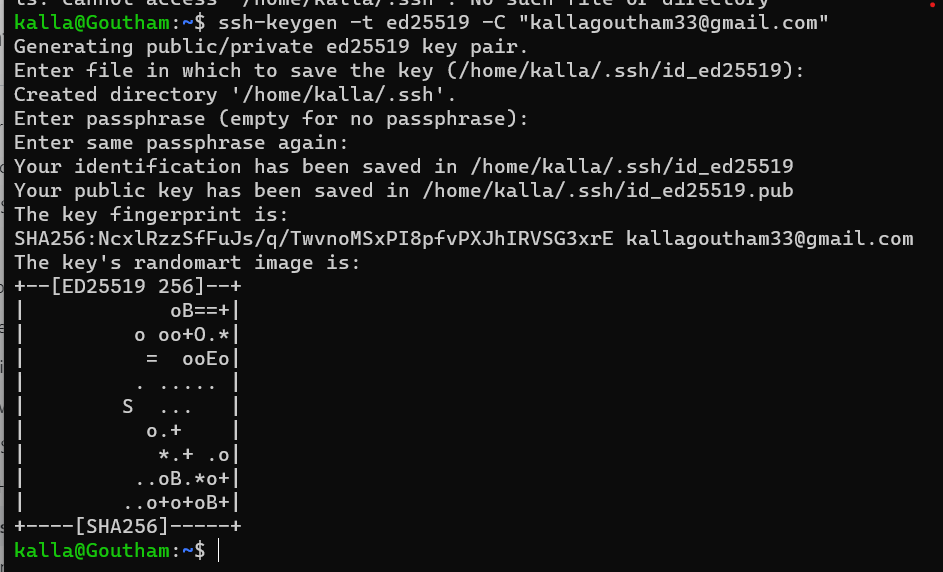
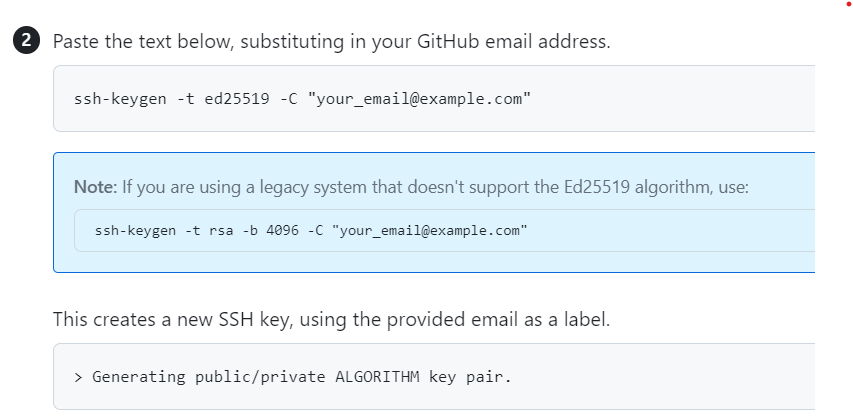
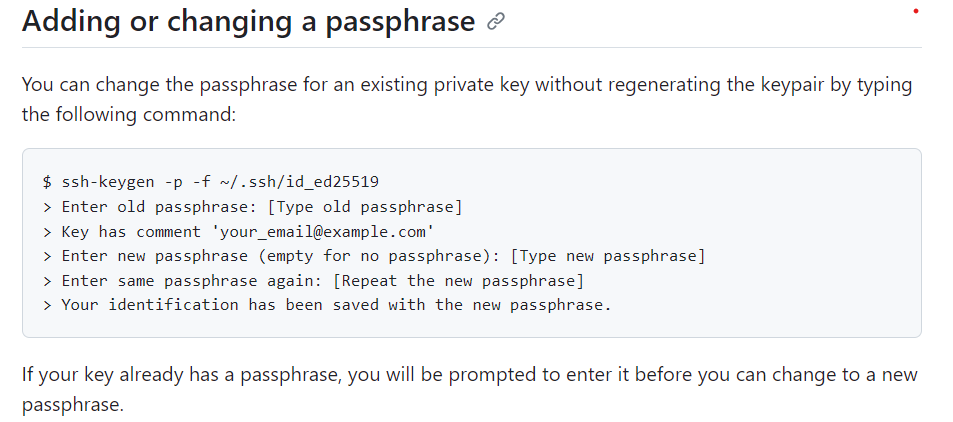
SSH is a protocol where the user has to be authenticated every time when an individual interacts with the git repo from the Terminal. We have to generate the SSH keys and intimate the GitHub about the SSH key credentials.

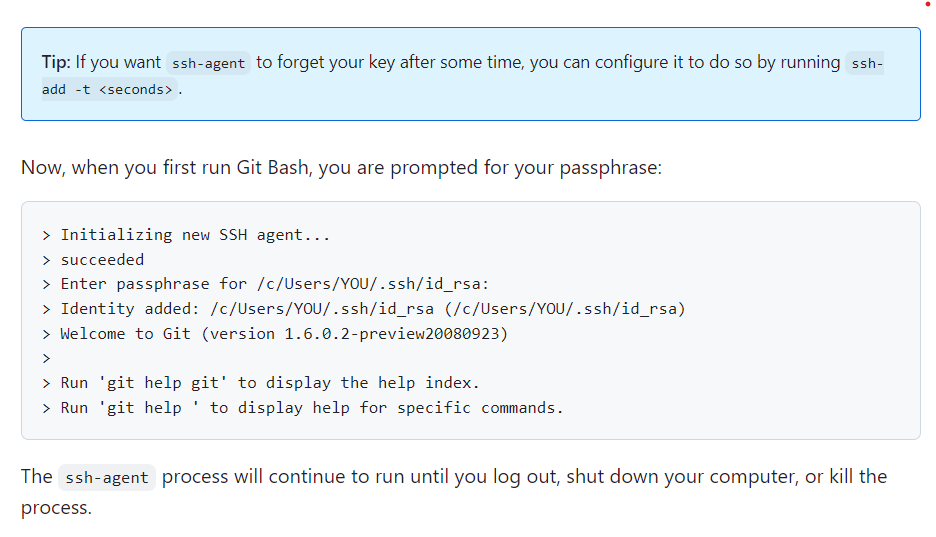
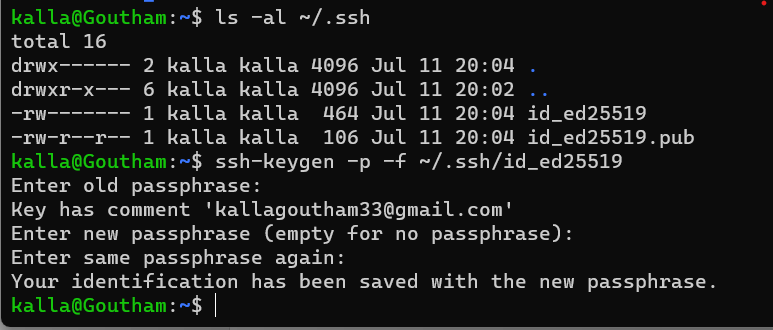
To check whether the ssh credentials present or not open the terminal and run the command

***ls -al ~/.ssh*** to check whether there are any ssh keys are present or not.

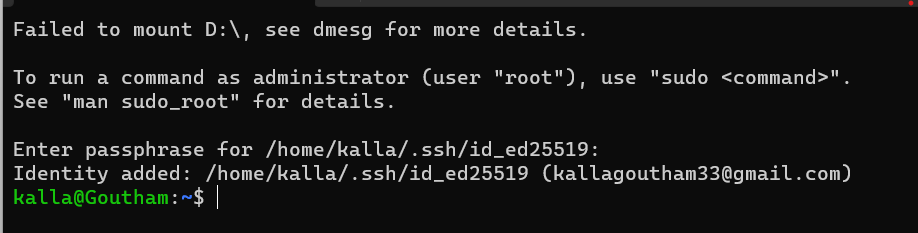


To Generate the SSH Keys:

1. Open Git Bash.
2. Now execute the following command in order to generate the new SSH keys for your windows.  If
3. When you are prompted to enter the file location where the key has to be stored press ENTER to save in the default file location. If the SSH key is also already generated it also prompts us to whether you want to rewrite the existing key or create a new one. Then the recommended way is to select the file location and give it a name.
4. Then you will be prompted to enter a ***passphrase .***Passphrase is an additional security added to your SSH key inorder to provide a layer of security where the user gaining cannot do anything without entering the passphrase. It is like a password to an account. 

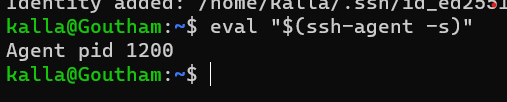


From now on when you open or call ssh-agent or run the Git Bash you are prompted to enter the passphrase so that the terminal gets open.



Now we have to add the SSH Key to the ssh-agent it can be done by running the below command

***eval “$(ssh-agent -s)”***

Now we will get the output and the SSH Key is added to ssh-agent. 

Now after verifying the SSH key added to the ssh-agent now you have to make sure to add the SSH private key to ssh-agent and to store the passphrase in the key chain. You have to run the command

***ssh-add ~/.ssh/id\_ed25519***(id\_ed25519can be replaced by file name or algorithm name)

***pbcopy < ~/.ssh/id\_ed25519.pub --------- for mac or windows***

***xclip ~/.ssh/id\_rsa.pub ---------for mac or windows***

***vim ~/.ssh/id\_rsa.pub -------- for mac or windows***

***clip ~/.ssh/id\_rsa.pub ------- for command line in git or windows***

The above command will display the SSH key now copy that to the clip board.

Now go to the GitHub profile go to settings now select ***SSH and CPG Keys*** and now set the name, Key type and paste the ssh key and click add. Now the key gets added to our git.

***How do the code from GitHub in our Local System:***

1. ***Existing Repo***

Now you have to add ***git init*** so that the git is initialized then add the

***git remote add <url>*** so that the existing repo and the folder in our local system gets connected then you can perform all the git actions so that it becomes easier

1. ***New Repo or from Scratch***

Then you have to create a repo and have to ***git clone <url>*** and make necessary changes and work on that and perform git actions on that.

Git Remote is the destination to the folder

***git remote -v or git remote*** 🡪 used to add list all the destinations added .

***git remote add <remote-name> <url>*** 🡪is used to add the remote to the git folder.

***git rename <old-remote-name> <new-remote-name>*** 🡪 used to rename the remotes.

***git remote remove <remote-name>*** 🡪 used to remove the remote.

***git push -u <remote> <branch>*** 🡪 used to push the code to remote and branch.

***git push -u <remote> <local-branch>: <remote-branch>*** 🡪pushes code from one branch from local to another branch in remote.

***-u is also referred as –set-upstream***

Whenever we work with remotes the remotes have all branches but our local doesn’t have all the branches.

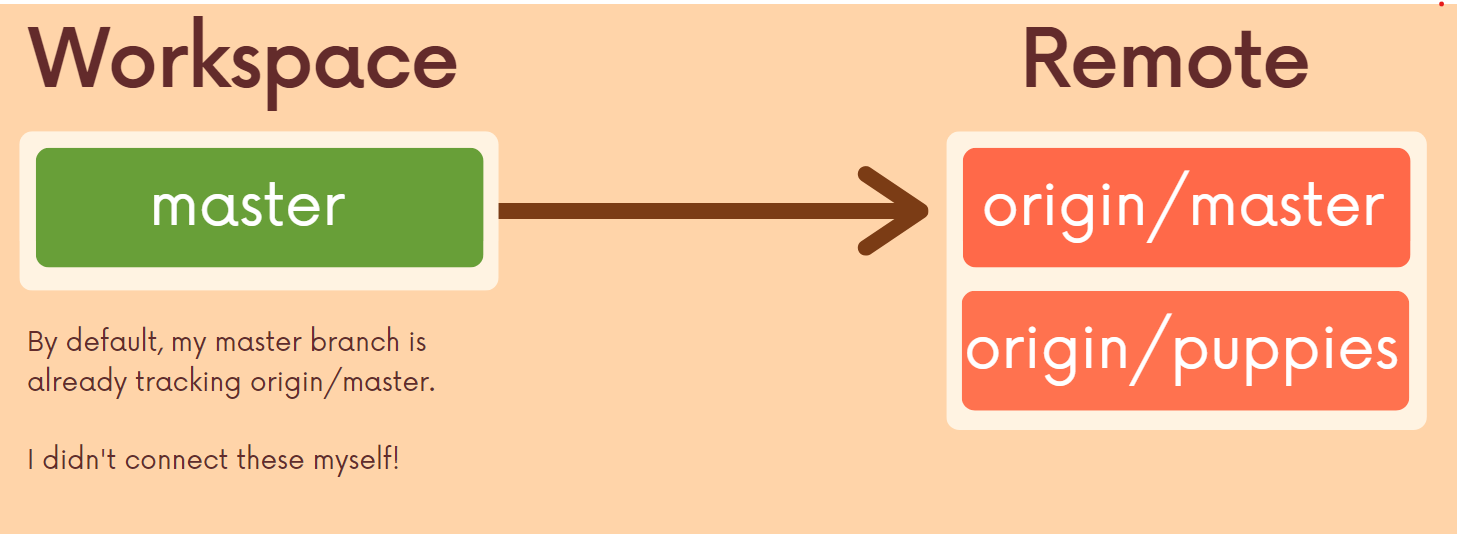
We have two branches with out git repo one is local HEAD for the branch and another is the remote HEAD of the git repo where the repo stands.

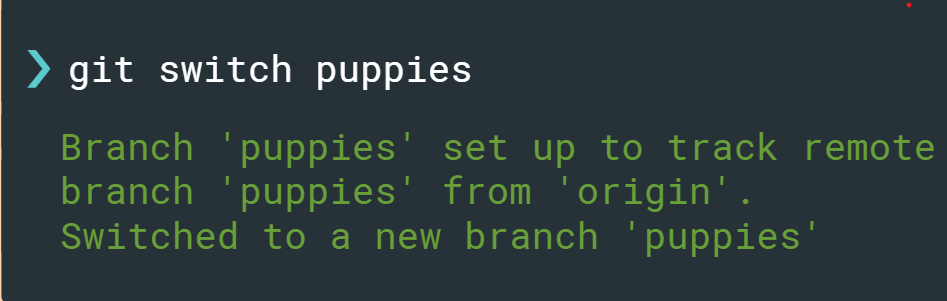
We have to make sure how the changes are getting tracked and we have to make necessary changes in the remote inorder to keep the repo up to date.

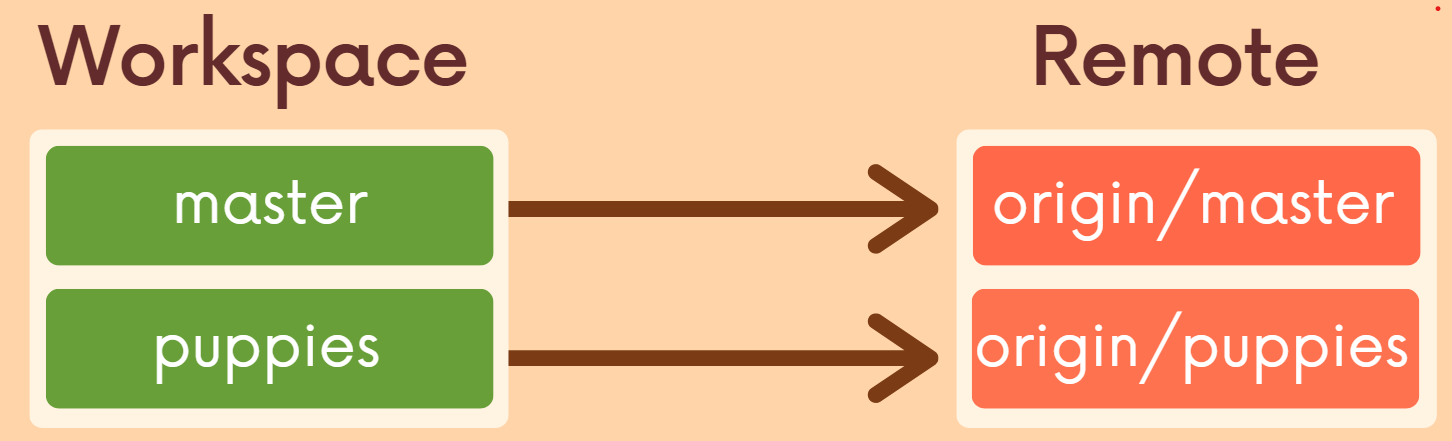
By default only the master branch is able to track itself with the remote master branch.

Now if we use git checkout origin/puppies then the git checkouts to that branch but it wont gets connected or sets the upstream.

If I use git switch puppies then it creates a branch in local machine name puppies and gets it connected to the remote branch puppies so that the upstream will get set.





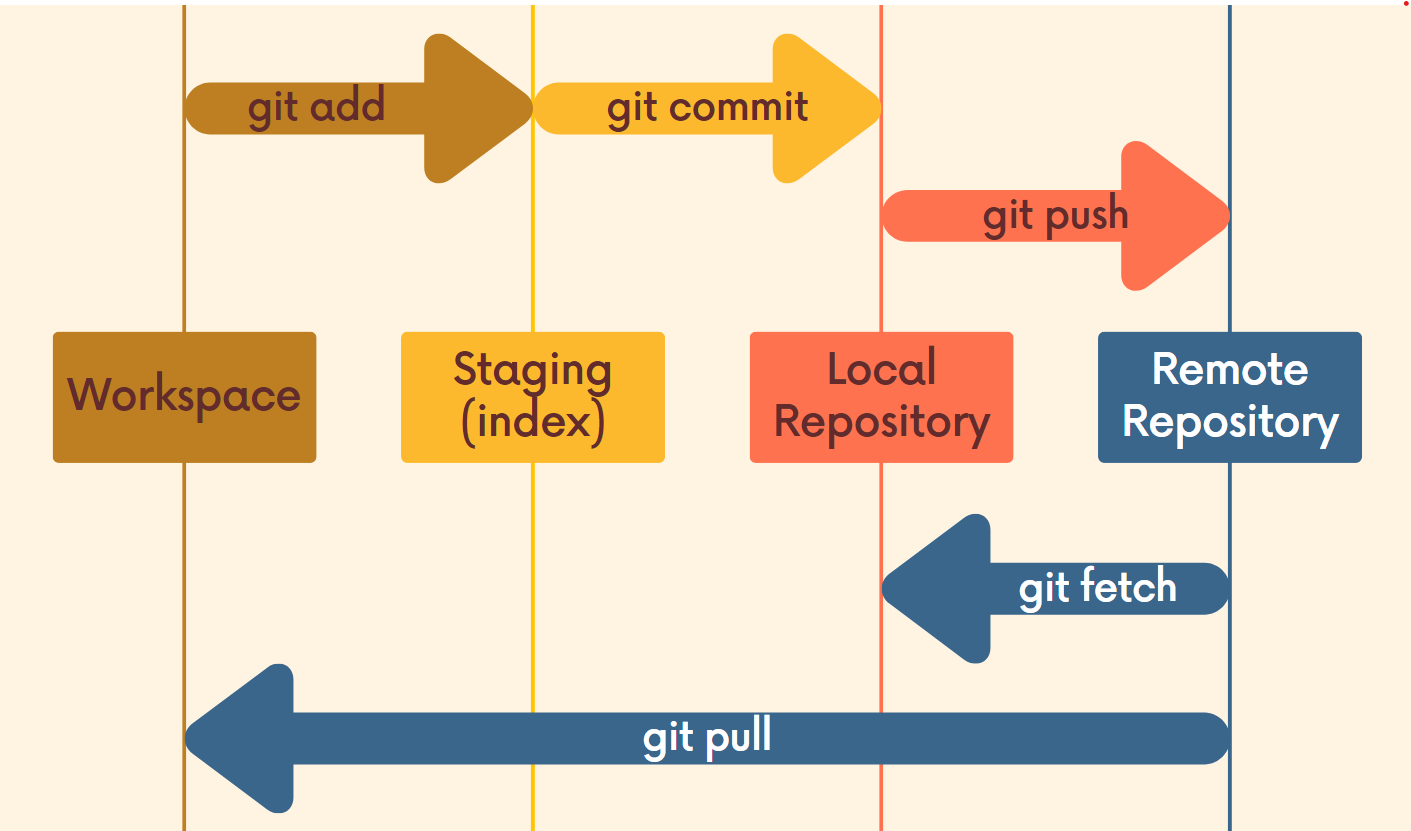


***git branch*** 🡪 tells on which branch we are right now.

***git branch -r*** 🡪 lists out all the remote branches present.

***git branch -a*** 🡪 shows out all the branches in the remote as well as local.

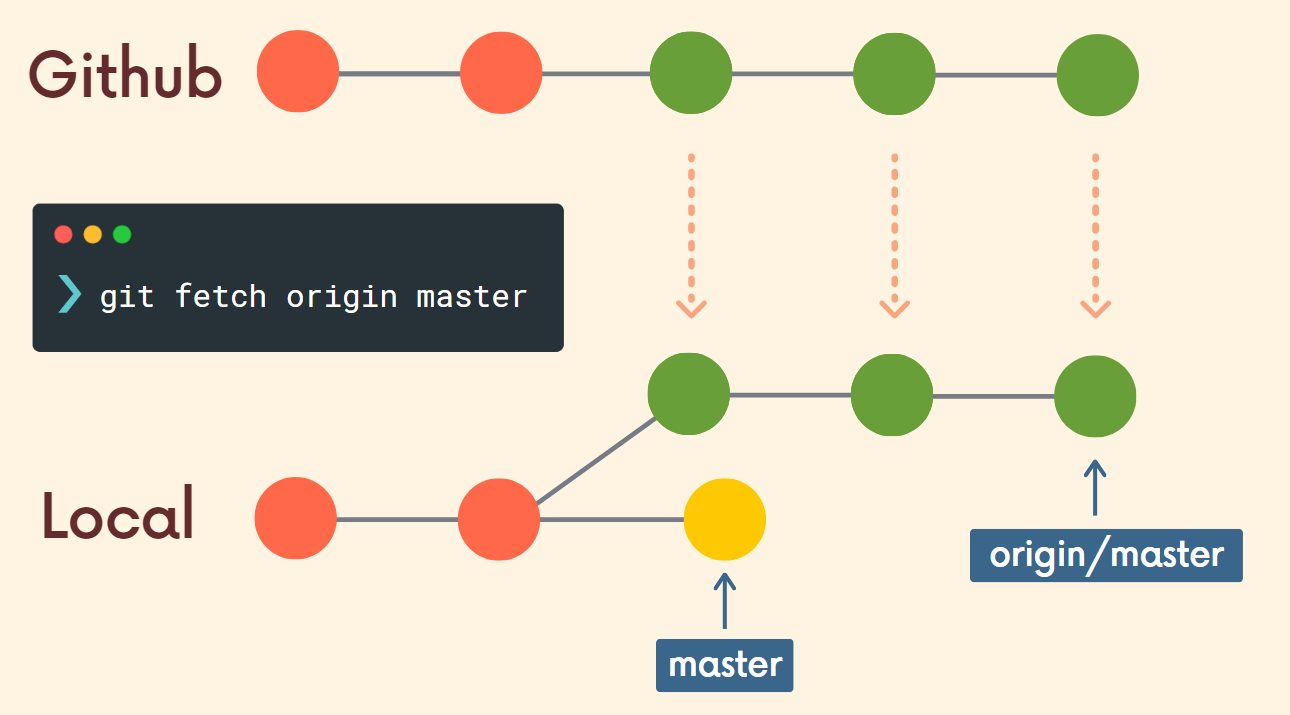
***Git Fetch and Git Pull***

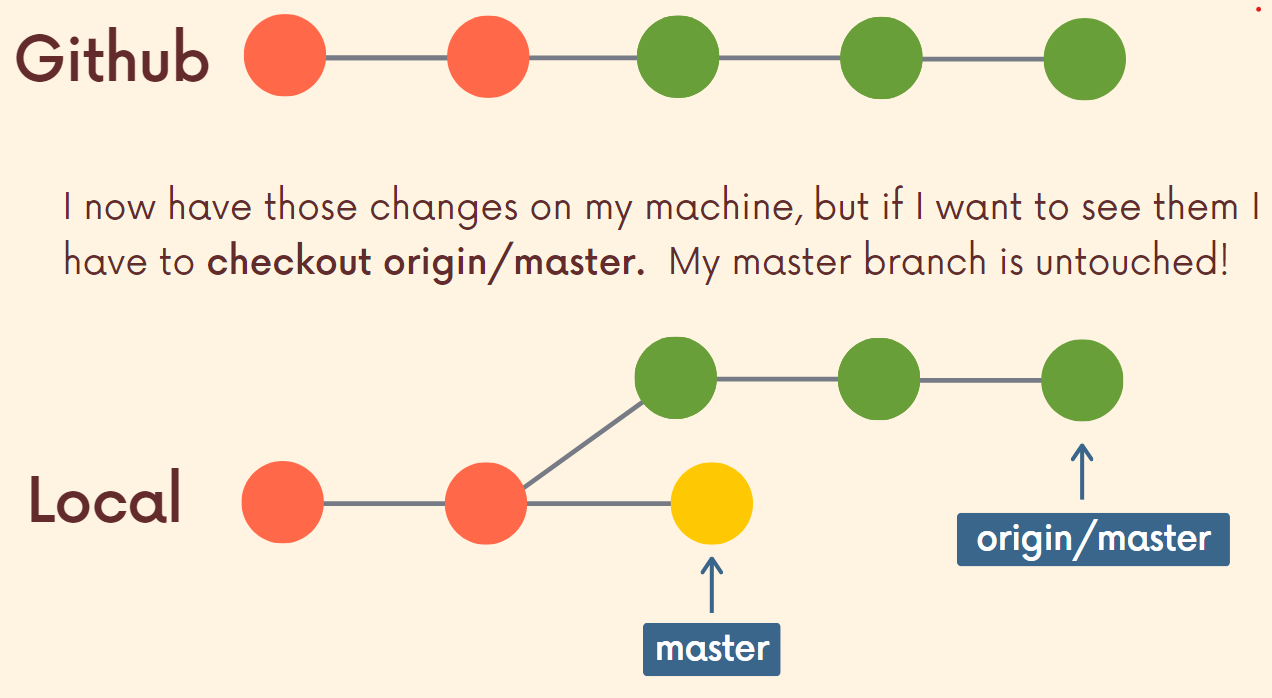
******

If we observe the above flow diagram we can clearly visualize the difference between the

***git fetch*** and ***git pull***. So we use ***git fetch*** to get changes to the Local Repository and we use

***git pull*** to get the changes to the WorkSpace.





***git fetch <remote>*** 🡪 used to fetch the changes from the remote to the local but not working space. Default remote will be origin and we can specify our origin.

***git fetch <remote> <branch>*** 🡪 used to fetch the changes from the specified remote and the branch from the git.

***git pull*** specifies the git to goto the remote and fetch the changes and add them or merge them with my current working directory.

***git pull = git fetch + git merge***

***git pull <remote> <branch>*** 🡪 used to pull the changes from the specified remote and branch from the repo.

\*\*Note: If we don’t specify the remote and branch in the git pull command then the remote is default set to the origin and branch is set to the branch on which we are currently working on.

***Public Repos*** : public repos are accessible and can be read by anyone on the internet and they are not able to push the code.

***Private Repos*** : private repos are cannot accessible by only the members who are granted permission.

***GitHub Collaborators*** : here we allow other to read or give direct access to a repo so that we can collaborate the code with different people / our team.

***README.md(markdown files)*** file used to communicate important information regarding the project.

Markdown is a text-to-HTML conversion tool for the web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then it will convert into the HTML easily.

We will write it in some standard syntax and the HTML will be produced automatically so that it is not required to write the HTML code for that it will automatically generate the HTML for that.

***Examples:***

***README file 🡪 HTML Tag Equivalent***

*# heading 🡪 <h1> heading </h1>*

*## heading 🡪 <h2>heading</h2>*

***\*\**** *This is Bold Text \*\* 🡪 <strong> This is Bold Text</strong>*

*\_\_This is Bold text\_\_ 🡪<strong>This is italic text </strong>*

***\**** *This is italic Text \*🡪 <i> This is Bold Text</i>*

*\_This is italic text\_ 🡪<i>This is italic text </i>*

*~~This is strike through text~~ 🡪 <del>This is strike through text</del>*

*Block quotes start with > , >> one > indicates one tab space.*

*Create a list by starting with +,\*,-*

*Sublists must have one more space extra than the parent list item.*

*Ordered lists are given by using the numbers.*

*`+` 🡪 inserts the + symbol in a box.*

*``` some code ``` 🡪 inserts a block of code to that. ```js* ***some js code*** *``` 🡪 highlights the js syntax and inserts a code block.\*

*Tables have some difficult syntax* |\_|\_|

|\_||\_|

*Links are rendered using [link text ](link url) 🡪 no space in between [] and ()*

*Images are rendered using ![alt in img](image url) 🡪 no space in between [] and ()*

***README.md*** file consists information like:

* What our project does.
* How to run the project.
* How it is different from others.
* How to install dependencies and update
* Is its net worthy and the source from where it originates.

***GitHub Gists:***

GitHub gists are a simple way to share code snippets and useful fragments with others. Gists are much easier to create, but offer far fewer features than GitHub Repository.

***GitHub Pages:***

GitHub pages are the public web pages that are hosted and published via GitHub. They allow you to create a website simply by pushing your code into GitHub.

GitHub pages are a hosting service for the Static web pages but not for the server side code like python, ruby, js but you can use HTML,CSS

There are two different type of GitHub pages. They are

* ***User Sites :*** They are given one per user means for every user of the github there will be one github page which is a UserSite the user site will look like

***username.github.io***  🡪 you can change it either.

* ***Project Sites :*** They are given one per every project(Repository) on the github the github page link will look like this.

***username.github.io/repo-name***  🡪 you can change it either.

After creating a Merge Request in GitHub we can merge the two branches using

***git fetch origin*** 🡪 fetches all the changes to the local machine.

***git switch <branch-to-be-merged>*** 🡪switches to the branch that has to be merged.

***git merge master*** 🡪merges the master branch with the branch that has to be merged with conflicts.

FIX CONFLICTS 🡪 RESOLVES CONFLICTS

***git switch master*** 🡪 switches back to master.

***git merge <branch-to-be-merged>*** 🡪 merges master with the branch that has to be merged now with no conflicts.

***git push origin master*** 🡪 push the code to the origin(Git Repo).

In Settings > Branches > Branch Protection Rules here we can add the Branch protection rules. In order to protect the branches from the illegal access by other resources.

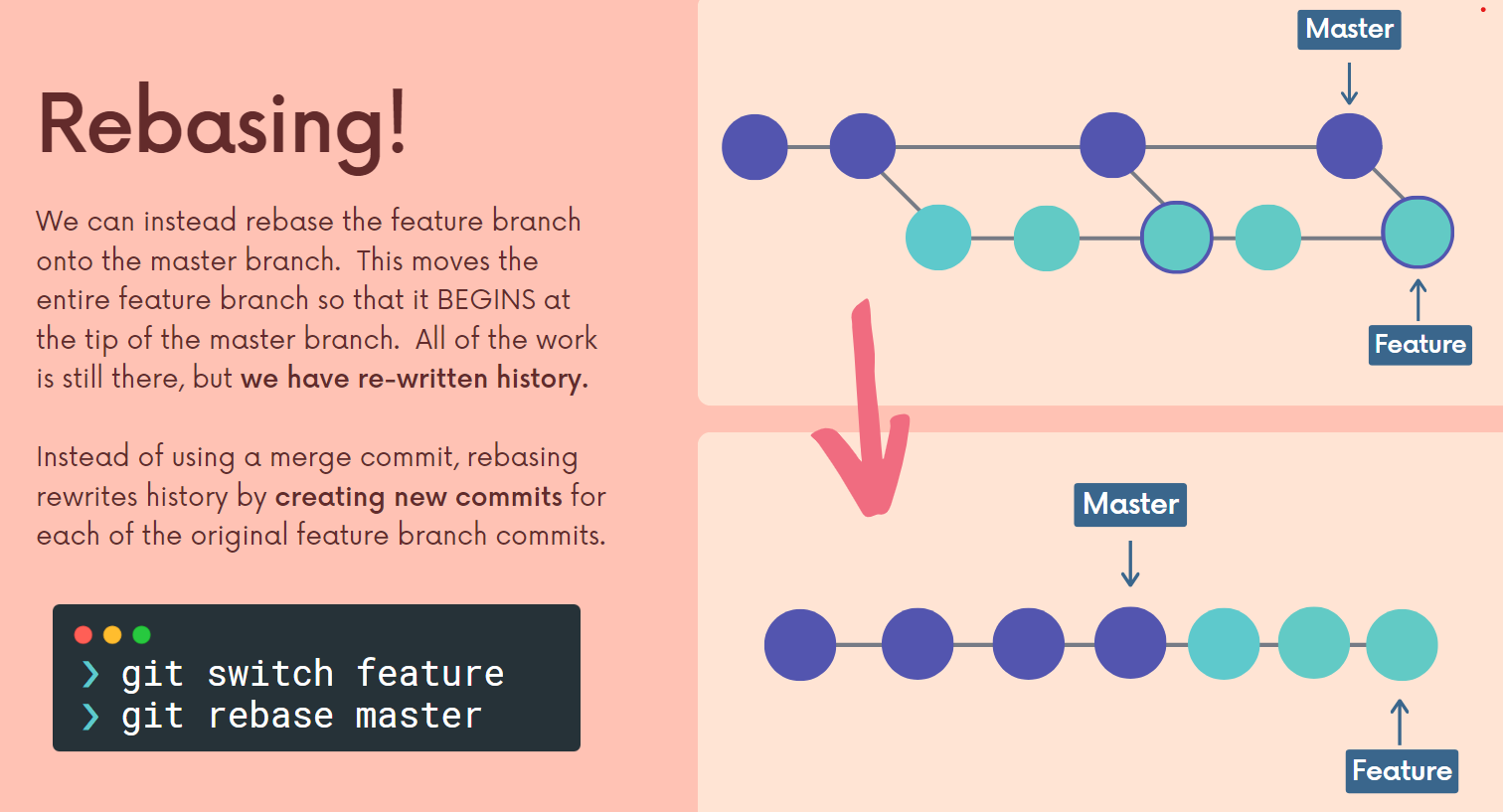
Whenever there is a large open source project then it is important to make fork and maintain our own repo / copy of source code where we make changes to our own repo before raising a merge request with the actual project. Fork is nothing but asking the hosting website to create a personal copy of code to me . We can fork a repo if and only if we have access to that repo.

***Git Rebasing:***

Git Rebase is also an alternative of git merging instead of git merge we can use git rebase.

Git Rebase can be used for two ways:

* An alternative for the git merge command.
* To clear the git log history.



Whenever we use Git Rebase we are actually rewriting the history of the git repo. Whenever we use git rebase instead of creating of mege commits it rewrites history and makes some new commits.

In the above diagram we can observe we are initially switching to the branch feature then we are rebasing the feature branch with the master branch it means that the we are rebasing and keeping all the commits from the feature into the Master and keeping it at the HEAD of the master.

***git rebase master*** on feature branch means it will keep feature branch commits on the tip of master.

To give nice history of git and how features are getting added to the very large open source project.

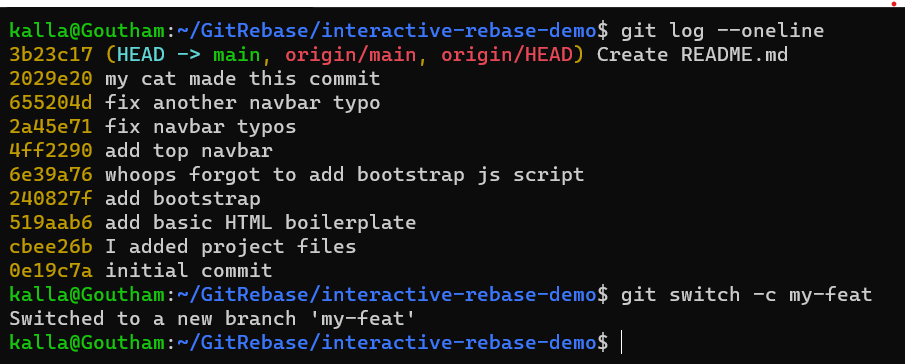
Here by using git rebase we are rewriting the HISTORY.

Similar to git merge we will also get conflicts by rebasing the branches also. Then we will resolve conflicts and we have to click ***git rebase --continue*** in order to continue to rebase.

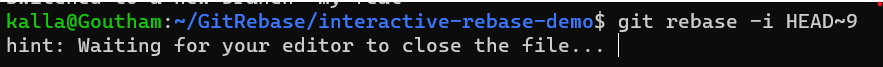
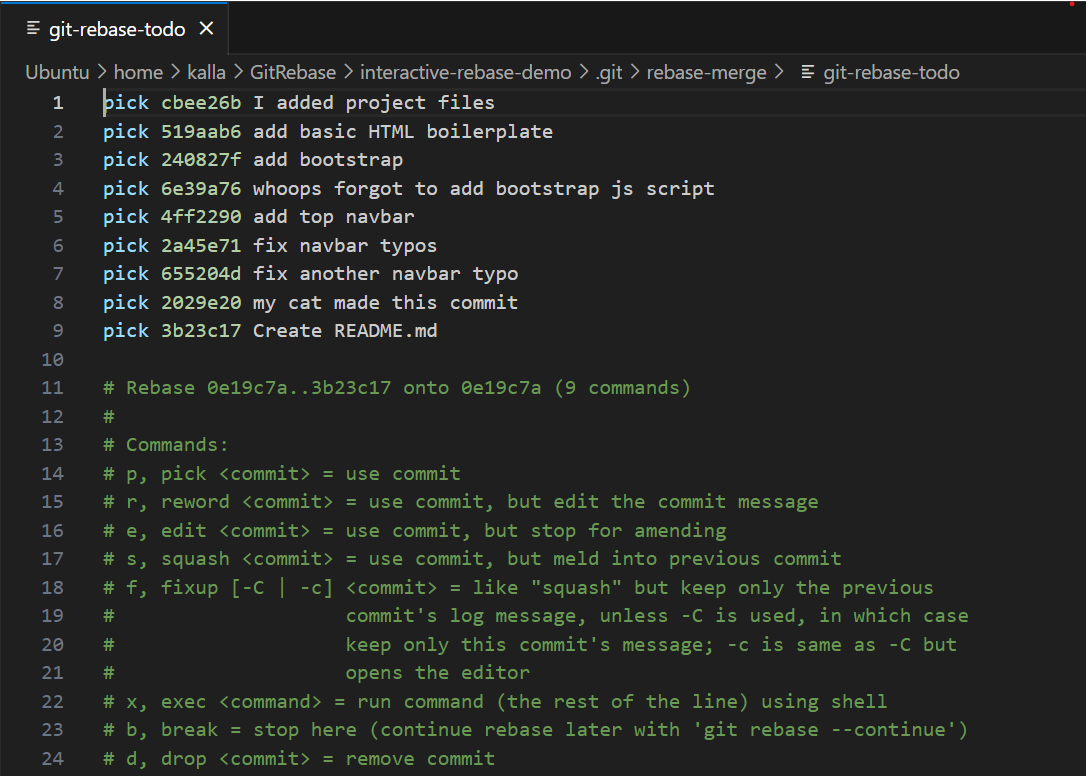
\*\*NOTE: NEVER REBASE BRANCHES WHENEVER THE COMMIT HISTORY IS IN GIT ONLY REBASE WHENEVER THE COMMIT HISTORY IS IN LOCAL MACHINE. BECAUSE IT MAKES MUDDIER YOU HAVING A DIFFERENT HISTORY AND YOUR COLLABORATOR HAVING DIFFERENT HISTORY.

***Interactive Rebase:***

Using Git Interactive Rebase we can rename, merge, remove the commits and clean up the log as we wish.



git rebase -i HEAD~n 🡪 takes the commit log until last n commits and it will be shown in the editor like this.

Those pick place is the place where the command is given to particular commit now on the commit messages actions are performed according to the actions. Please check the actions above mentioned.

***Git Tags:***

Tags are pointers to the particular commit in the Git. They are used to save a point they are like check points. In general tags are used to maintain versions. All the tag names must be unique.

There are two types of tags:

* ***Lightweight Tags:***

They are light weight tags which are just a name/ label to a particular point / commit on the git repo.

* ***Annotated Tags:***

Store extra meta data including authors name, email, the date, the tagginIg message.

Versioning is a method in which we assign give a version numbers to the releases of codes.

***Semantic Versioning*** is a type where the version numbers consists of three periods.

The ***x.y.z – <something>*** is how the version looks like here

x🡪 major release (whole new features are added

y🡪 minor release

z🡪 patch release

something 🡪 may be like beta , alpha , beta 2.0 etc..,

For example : 2.4.1

***git tag*** 🡪 used to list all the tags in the git repo.

***git tag -l “<regex>”*** 🡪 used to list all the tags which matches the given regex.

***git checkout <tag>*** 🡪 used to jump to the tag now the branch will move into detached HEAD in order to jump to particular commit.

***git diff <tag1> <tag2>*** 🡪 used to compare the changes / differences between the two tags.

***git tag <tag-name>*** 🡪 creates a light weight tag at the current position of HEAD with the tag-name.

***git tag -a <tag-name>*** 🡪 creates a annotated tag at the current position of HEAD with the tag-name.

***git tag -a <tag-name> -m “message”*** 🡪 annotated tag with message.

***git show <tag-name>*** 🡪 used to see more information about the tag.

***git tag <tag-name> <commit-hash>*** 🡪 tags the previous commit with tag-name which points to the given commit-hash.

***git tag -d <tag-name>*** 🡪 used to delete the tag which is referred to tag-name.

whenever we push code into the remote repo it wont push up the tags if we want to push tags we have to run some separate commands.

***git push --tags*** 🡪pushes all the tags that are present.

***git push <remote> <tag-name>*** 🡪 pushes the tag with tag-name to the specified repo.

***config*** file in ***.git*** folder is used to set the configuration for the git repo, or across machine, or to set the config globally.

Inside the ***refs*** folder the references are stored. In general references are the commit hash which points to a particular commit.

In refs/heads contains as many files as many as our git repo branches. Every file name is branch name and the contents is the tip / latest commit-hash on the particular branch.

In similar fashion refs/tags all tags are the files and the contents are the commits.

In refs/remotes all the references to the remotes are stored here.

The ***HEAD*** file in the ***.git*** folder consists of the ***commit-hash*** to which the current HEAD points.

***objects*** folder in the ***.git*** is the backup is stored, here only the history is stored, the content is in binary format. 4 types of git objects are commit, tree, blob, annotated tags. All this history is saved in this folder.

HASHING FUNCTIONS are functions that map some input data of some arbitary size and give output of fixed size.

Git uses the SHA-1 algorithm for the hashing SHA -1 generates the 40-digit hexadecimal output.

Git uses a ***key-value data store***.

Git stores all the data like folders or files in the form of the value and key will be the hash of the particular information. If we ask for something then it hashes it compares and gives the corresponding value of the particular key/hash.

***git hash-object <file/data>*** 🡪 gives the unique SHA-1 hash of the specified file/data.

***echo “hi” | git hash-object --stdin*** 🡪stdin means the echo “hi” and | indicates the pipelining. Here it gives the SHA-1 hash for the ***echo “hi”***

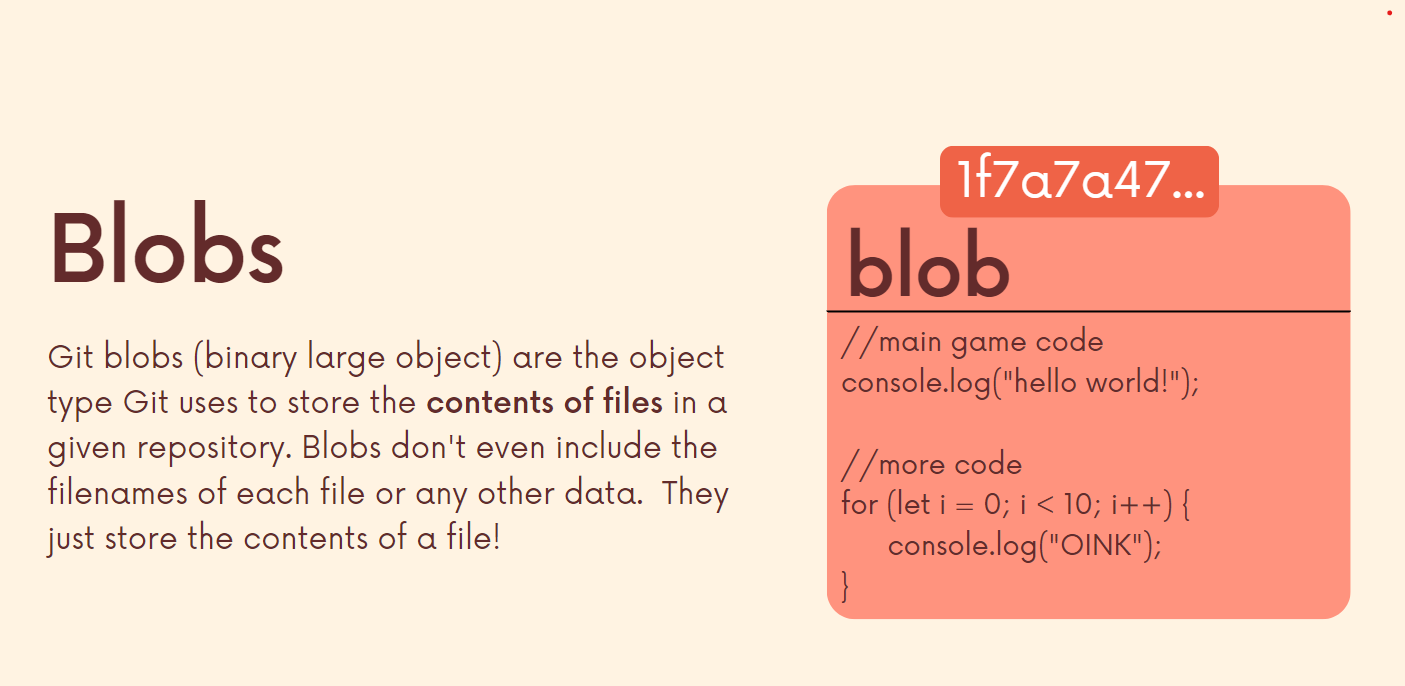
***echo “hi” | git hash-object –stdin -w 🡪***specifies git to store the hash in the objects.

***git cat-file -p <sha-1 hash>*** 🡪 gives the output.

***git cat-file -p <branch-name>^{tree}*** 🡪 here its displays the tree structure that is tip of the specified branch.

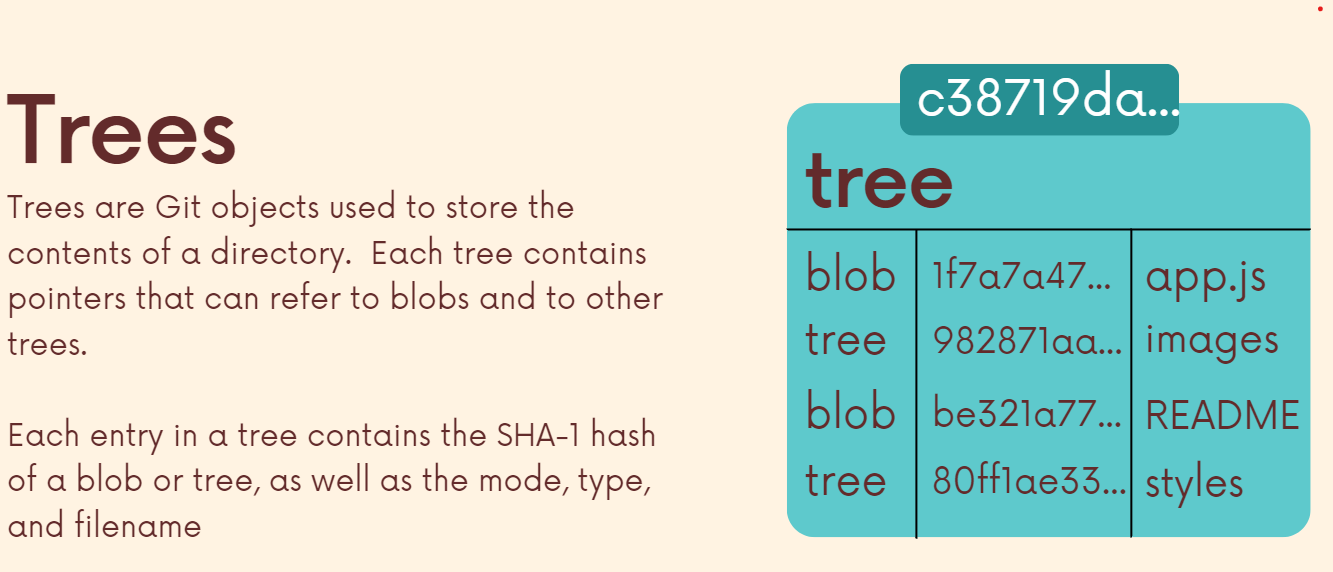
***git cat-file -t <sha-1 hash>*** 🡪 specifies the given hash belongs to the tree or a blob.

Git blobs are binary large objects that git uses to store the contents of the file in a given repository. Blobs doesn’t even include the filename of each file or any other data. They just store the contents of the file.



Git Trees are objects that store the contents of a directory. Each tree contains the pointers that can refer to the blobs and other trees.

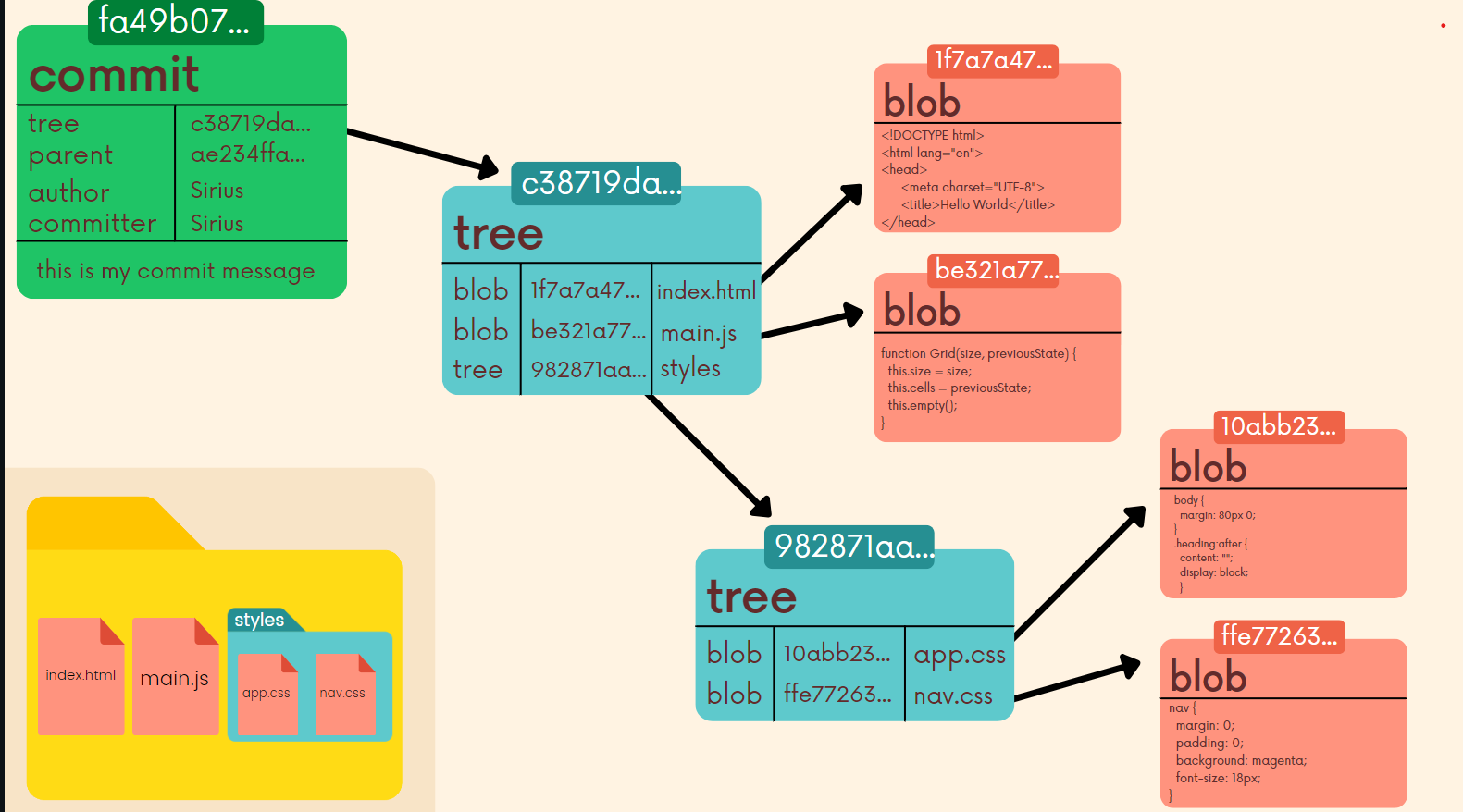
Each entry in the tree consists of the SHA-1 hash of the a blob or a tree as well as the mode and filename.





***Git commit*** is an object which consists of the reference tree and reference parent commit and the commit message and the author and the committer. Here every time the previous commit becomes the parent of the current commit. The commit generates a sequence of linked list like structure.

For every commit one tree is generated during the commit.



Here for every commit the trees getting updated and will add to the next commit and so on.

***Git Reflogs:***

Git Reflogs are also known as reference logs. In git file system it is important to keep track of many references like branch references, detachd HEAD references, HEAD, merge HEAD, diff HEAD etc., The git reflog will store the data and used to retreive the lost data.

In the ***.git*** folder there will be a directory known as ***logs*** and in the logs there will be a file HEAD which is readable and it keep tracks the HEAD over all the history in our project.

The limitation of reflogs is that it is local and they store only the entry/ information of current local changes and they do expire they expire every 90 days and it can be configured the way we want.

The git reflog commands accepts the sub commands of show, expire, delete but generally expire and delete are not used the only most widely used reflog is git reflog show

***git reflog show*** 🡪 shows the reflog with reference to HEAD as the default is HEAD.

***git reflog show HEAD*** 🡪 shows the reflog of HEAD.

***git reflog show <branch-name>*** 🡪 shows the reflog of the main.

***Reflog References :***

If we see the reflog of any git repo we can observe the reflog references look similar like this ***name@{qualifer}***. Like HEAD@{2} means go to the step 2 back in the reflog history.

We can use the reflog references using timed references also like

***git reflog*** [***master@{one.week.ago}***](mailto:master@%7bone.week.ago%7d)

***git reflog*** [***bugfix@{2.days.ago}***](mailto:bugfix@%7b2.days.ago%7d)

***git diff main@{0} main@{yesterday}***

In this way we can also use the timed references in the git reflogs. In this way we can also get the history on the time basis.



***git reset --hard <commit-hash>*** 🡪 resets the current working directory to the specifies commit hash.

If we reset our current directory if we are going to detached HEAD we cant see the commits in the git log but the hash and the commit is present in the reflog. In this way it is used to undo the hard reset.

***git reset --hard master@{1}*** 🡪 undoes the hard reset performed in the master branch.

In the similar way the reflog can also undo the git rebase also.

***Git Aliases:***

Everytime the git looks at the global config file or the local config file. ***~/.gitconfig*** to pick all the settings. The config variables are set from that file.

We can add git aliases very easily for example like below in the .gitconfig file :

***[alias]***

***s = status***

***l = log***

here s means status so you can use instead ***git s*** is same as ***git status***

means here alias refers to ***git ci instead of git*** commit in this we can set our custom aliases for the commands.

We can set the alias not only by editing the .gitconfig file but also using command line.

***git config --global alias.showmebranches branch*** 🡪 now the ***git branch*** is same as ***git showmebranches***.

In this we can also set aliases from the command line.

\*\*Note : We can also the arguments using the git aliases also.

Some useful git aliases :

<https://www.durdn.com/blog/2012/11/22/must-have-git-aliases-advanced-examples/>

https://github.com/GitAlias/gitalias