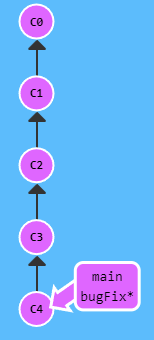
1. Branch

* Flags

1. -f <branchName> <commit/branchName>

Branch forcing,

For ex.:



git branch -f main C2

1. -M

Basically renames the current branch to the given name, and if the target name branch already exists, it overwrites it.

git branch -M <newname>

//or

git branch -M <oldbranchname> <newname>

The difference is that the first one is already checked out on current branch so it remains checked out on the new name branch too, but in the second one whilst it could rename any branch, it won’t check it out.

1. -u

Changes the tracked remote branch for a local branch.

git branch -u <remote branch> <local branch>

Sets the given local branch to track the given remote branch, if we omit local branch name and are checked out on any branch then that branch’s remote tracker is updated.

1. Checkout

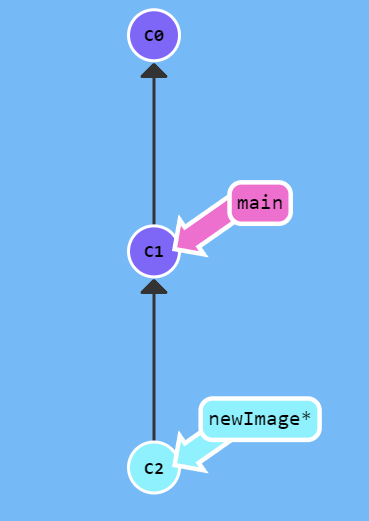
Checkout basically tells git to prepare to local repository as it should be on the given commit.

git checkout <branch/commit>

checks out the given branch and HEAD is now at attached to this branch.

If we checkout a commit (by giving its hash) instead then that commit is checked out and HEAD is placed at the commit. We don’t need to provide the whole hash of the commit, just enough characters from the start till it can identify the commit uniquely.

For ex.:



git checkout newImage brings local repo to C2, and git checkout C1 brings local repo and HEAD to C1.

Doing git checkout when the local repo has uncommited changes will present either the option to stash or discard the changes, stash meaning they are saved and can be reapplied later.

* Flags

1. -b <branchName>

Create a new branch and checkout it.

git checkout -b Yo

Creates a branch Yo at HEAD and checks it out.

1. Remote Tracking

We can set a branch’s tracked remote branch using

git checkout <branchName> <remote branch to track>

For ex.:

git checkout Yoo o/main

checks out Yoo branch and sets its remote target o/main.

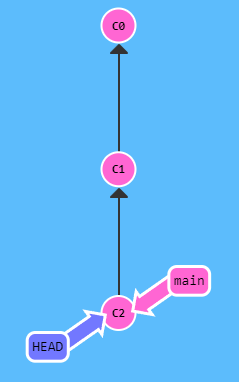


1. HEAD

Just like a branch is a pointer to a commit, a HEAD is a unique pointer that only exists in local git commit repo that points to the current checked out commit. Normally HEAD is attached, i.e., it points to the current checked out branch’s latest commit, so HEAD is pointing to the same commit as the branch and is attached to it, so it moves with the branch pointer.

But if we checkout any given commit by its hash instead, then the HEAD points to that commit and it is now in detached mode.

For ex.:



git checkout C2 here brings HEAD into detached mode, even if we were already checked out to main which is at C2 and HEAD was already at C2. It also checks out C2.

1. Switch

git switch is a replacement for checkout and was introduced in v2.23.

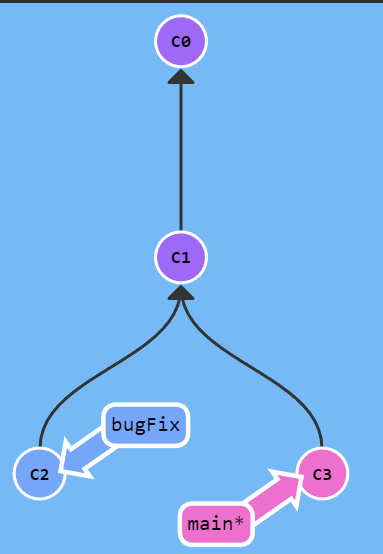
1. Merge

Creates a new commit which merges 2 commits, the commit created by merge has 2 parents. Basically merges 2 commit chains together. The new commit is created at the current HEAD.

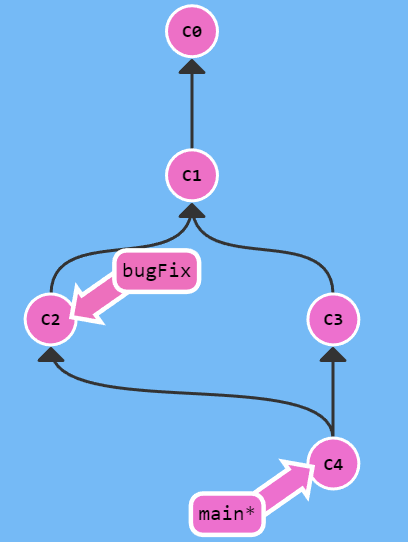
git merge <branchName>

Creates a new commit at HEAD which has <branchName>’s latest commit merged into it. Then moves the branch to the merged commit, and checks it out.

For ex.:



Here HEAD is at main, so if we do git merge bugFix it will do



Now if we checkout bugFix and merge main in to it, it will point to C4 too, git doesn’t create a new commit here.

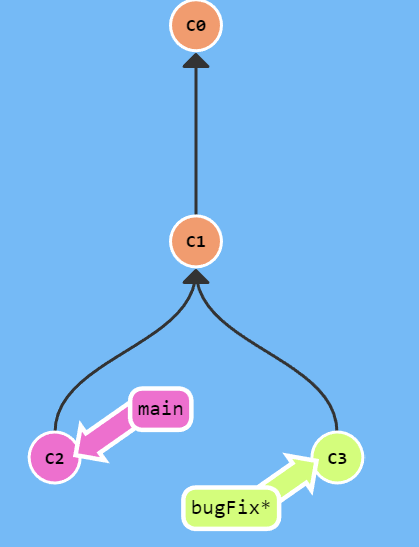
1. Rebase

Rebase is an alternative to merge, it doesn’t create just 1 merged commit, instead it takes the commits from the history of HEAD not in the target branch and then sequentially (by commit create time) adds them to the chain of the target branch.

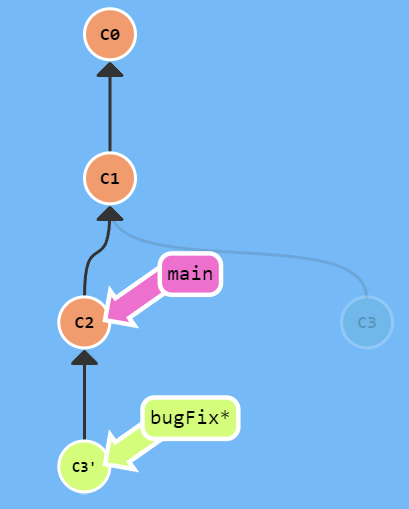
git rebase <branchName>

Finds the commits in the current HEAD’s history not in <branchName>’s history and adds them sequentially after the latest commit of the branch then puts the current HEAD/branch at the latest commit in this chain and checks it out.

For ex.:



Here, we are at bugFix, and now if we do git rebase main, it will do so



And similar to merge, if we checkout main now and rebase it onto bugFix it will move main to C3’ instead of creating another new commit.

The benefit to rebase is that it preserves the commit history of the current branch and makes the history contain all commits not just the latest commit like merge.

* Flags

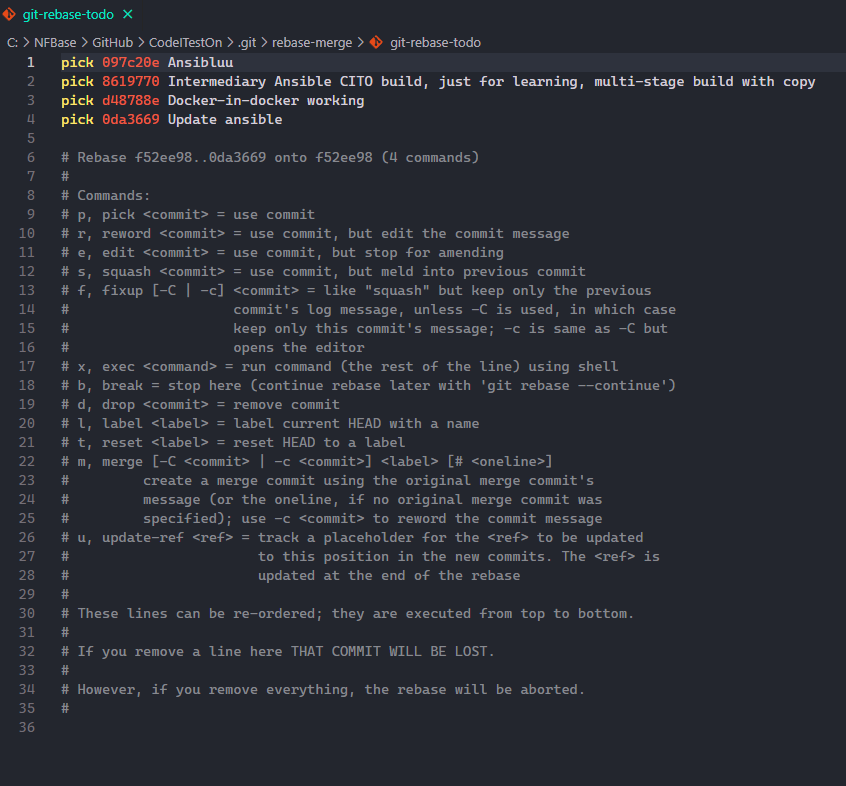
1. -i <commit>

This is just like git cherry-pick except it selects multiple commits between current HEAD and given commit and then also presents an interface (generally a text editor) with the commits to select/reorder etc. and then rebasing over the given commit.

The major difference between interactive rebase and cherry-pick is that -i does not apply the commits just below the current HEAD, it applies the commits after the given commit as it rebases HEAD/branch over that commit.

For ex.:

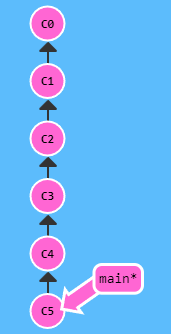
git rebase -i HEAD~4 in my git repo presented this



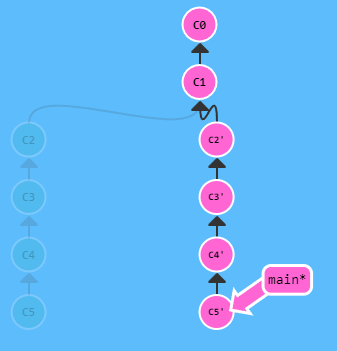
Here I can select which commits to pick, not pick, and most importantly order them. Currently it says, pick 097…, then 861… and so on. And finally apply 0da… , then apply these commits in this order after 097… and move the current HEAD/branch to 0da.. and finally check it out. We can change this order by simply changing their order in this opened text editor.

Finally, closing the editor will allow git to apply the changes.

Another example:



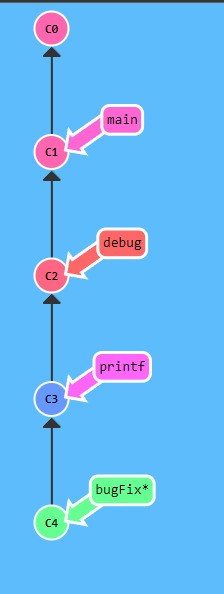
git rebase -i HEAD~4 will do this



considering all the commits were picked in the order as-is.

So main has been rebased over HEAD~4, which is C1 and brought commits between it’s previous position that is C5 and C1.

-i also allows us to rebase over a commit already in history, for ex.:

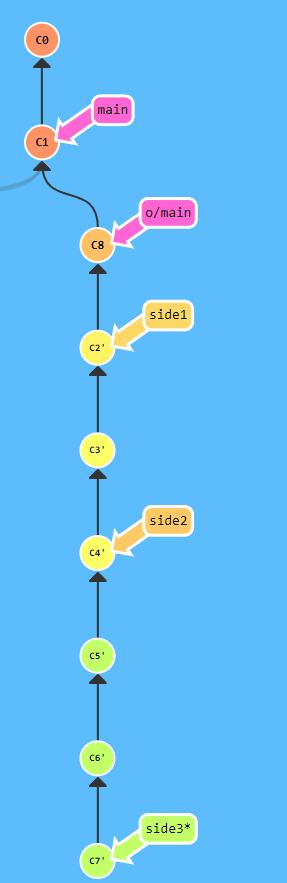


git rebase main here will fail as C1 is already in the history of bugFix, but if we do git rebase -i main and only select C1, then that will work.

* Fast Forwarding

If we have a commits after a given branch, then we can move the branch to any child commit using rebase. This is called fast forwarding.

For ex.:



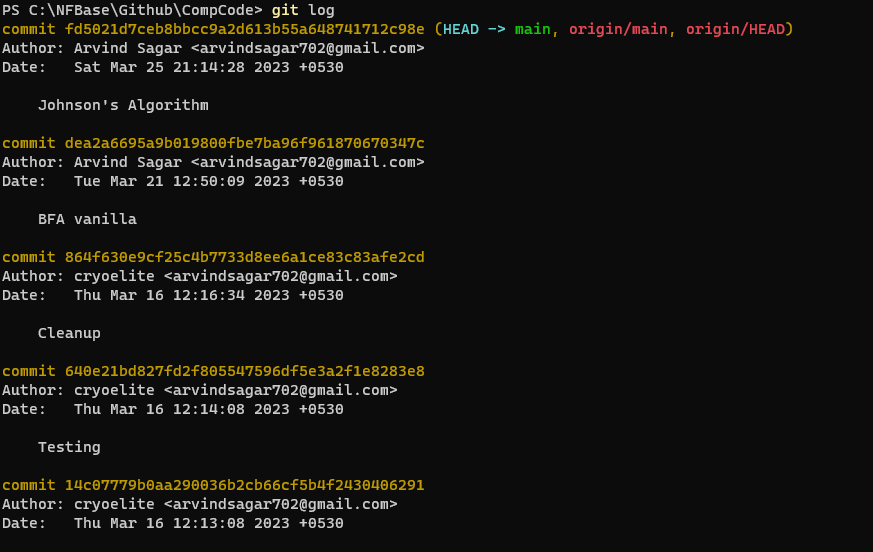
Here if we do git rebase side3 main, so rebase main on side3, then main will jump to C7’ and be checked out. This checkout won’t affect the local repo as we are still on the same commit in this case.

1. Log

git log

shows the commit history.

For ex.:



Shows me the commit history for the current branch, that is main. It also shows HEAD is attached to main. If there are a lot of commits, they will be paged and pressing enter will scroll while typing q and then pressing enter will quit the command.

* Flags

1. <since>..<until>

We can find the commits on one branch and not on the other with these args.

For ex.:

git log origin/main..main

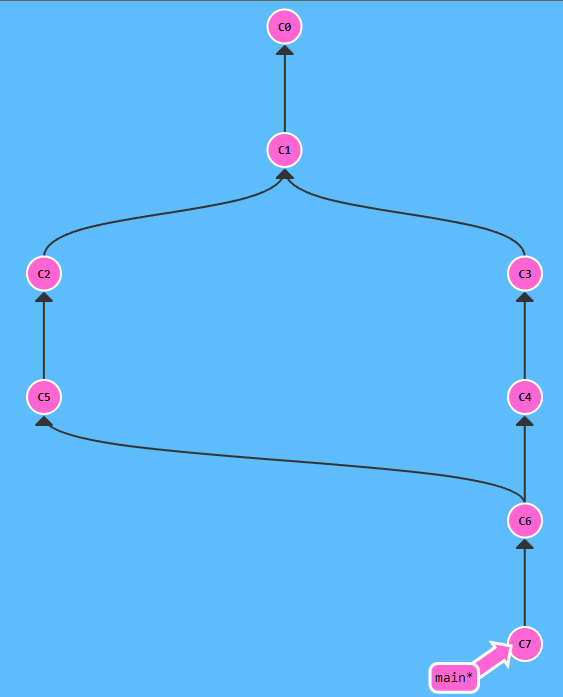
will show all the commits in main not in origin/main, or in other words, local commits not on remote repository yet.

1. --first-parent

By default log shows all commits going up the commit history, but if we wish to know only the commits in this branch and not the commits (with their histories) that got merged into this branch then we use this option.

The first parent of any commit is the commit that was merged into by another commit.

For ex.:



Here for C6, if C5 was merged into C4 then C4 is the first parent and git log --first-parent from C6 would give C4, C3, C1 and C0.

1. show

git show

A simpler version of log, only shows the last commit’s details, including the difference between last commit and current commit.

* Flags

1. --first-parent

Just like log’s first parent, if a commit has 2 parents, then it shows the first parent commit.

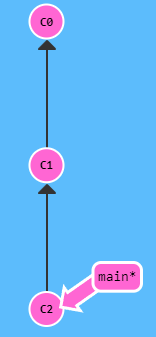
1. Ancestry/Relative Reference

For faster navigation over commits, git allows us to use relative references instead, they can be used in place of commits for any command that accepts a commit hash. A relative reference tells git to look for a commit given integers before the given commit.

There’s 2 of these,

^ and ~<num>.

For ex.:



git checkout main^

will checkout 1 commit before main (C2) so it will checkout C1.

We will get a detached head with this as we are checking out a commit.

Similarly,

git checkout main^^

//or

git checkout main~2

will checkout C0.

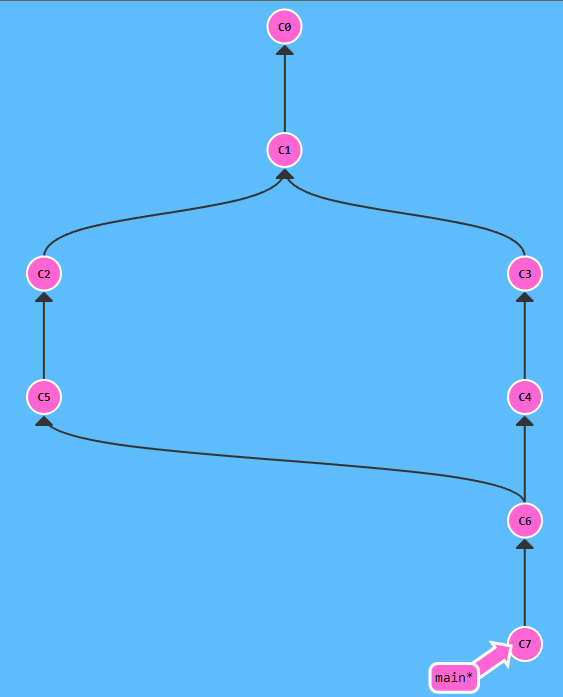
We can also specify a commit instead, so git checkout C2^ will checkout C1.

And obviously we can use HEAD to base off relative reference too.

* ^ has another format, ^<1 or 2>

This specifies the parent # to be taken. By default if we omit this int then the value is 1, which means the first parent. ^2 means the second parent.

For ex.:



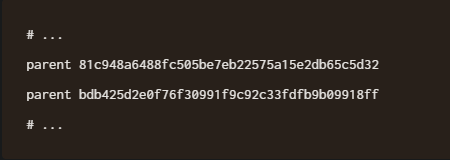
Here C6 has 2 parents, to know which is the first one, we can either do

git log --first-parent

//or

git show --first-parent

If we omit --first-parent then shows something like



Say C5 is the second parent.

Now, to refer to either parent using relative reference we can use ^<1 or 2>

For example:

git checkout C6^2

checks out C5.

It can be mixed with ~ and multiple ^^.

So,

git checkout C7^^2~2

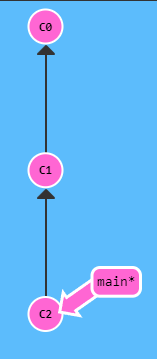
checks out C1 going C7, C6, C5, C4 and C1.



1. Reset

Git reset makes a branch point to a previous commit in its history, essentially undoing any of the commits from the latest commit to that commit.

For ex.:



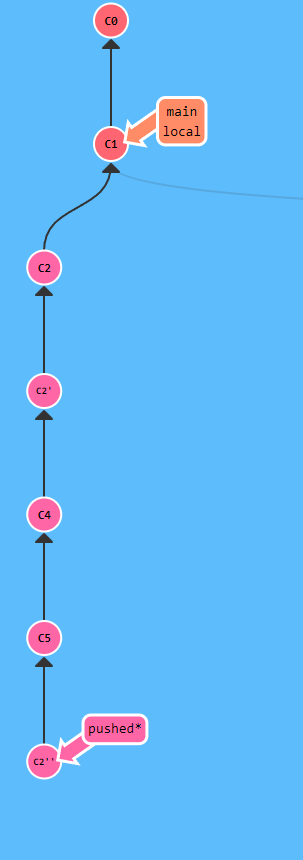
git reset HEAD^

here will make main point to C1.

1. Revert

Unlike Reset, revert creates a new commit after the latest commit with the previous commits undone.

For ex.:



git revert HEAD~3

on ‘pushed’ branch at C5 simply created a single commit with C2’, C4 and C5 undone.

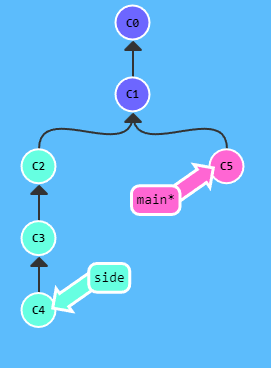
Reset can only be used on local commit history that hasn’t been pushed as it essentially hides the later commits. On remote, we can only use revert.

1. cherry-pick

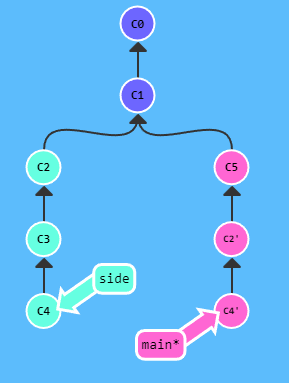
Picks the given commits and applies them after the current HEAD.

git cherry-pick <commit 1> <optional commit 2> <optional commit 3> …

For ex.:



git cherry-pick C2 C4 will add C2 and C4 after main/HEAD and then check out C4 like so

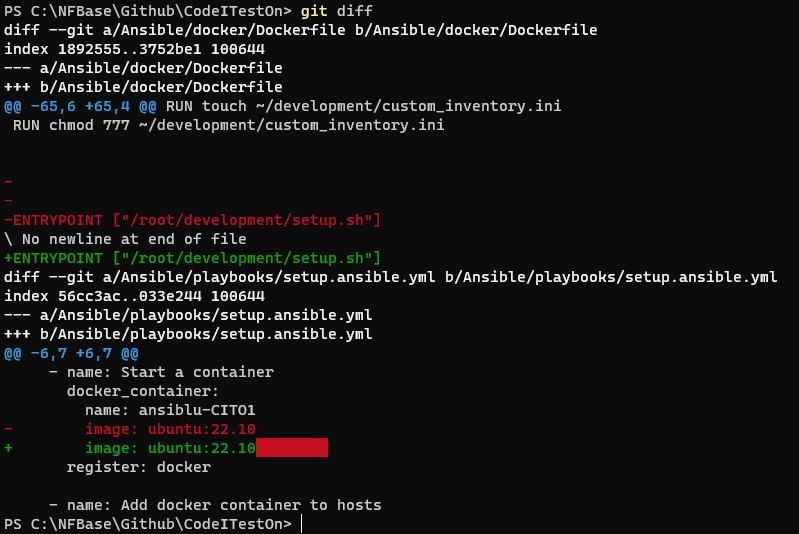


1. diff

This arg for git shows the difference between the last commit of this branch and current local repo.

git diff

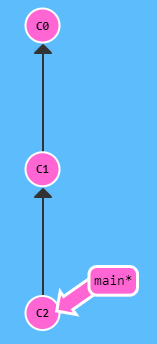
For ex.



1. Tags

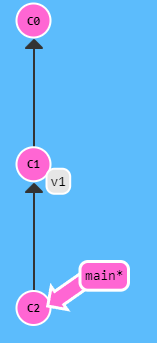
Git Tags allow special labels to be assigned to commits to mark the commit. This doesn’t affect the versioning system at all and is only for semantic purposes, such as marking a given commit (with it’s history) as a major release and then tracking all major releases through git tags in some other system, such as in Microsoft Azure platform. Git Tags are permanent and are immutable, meaning if they attach to a commit then they are never going to give a different commit whenever referenced, yes they can be used to navigate around the commit graph.

For ex.:



git tag v1 C1

will add a tag “v1” to the commit C1.



The commit hash is optional, if we don’t specify it then the tag is attached to the HEAD’s commit.

We can’t checkout a tag like we do with branches, because it doesn’t belong to any branch and in checking out a tag, we go into detached HEAD mode much like checking out a commit directly. In essence, a tag is just an alias for a commit’s hash so instead of specifying a commit hash we can specify a tag if the commit has one and it will reference that commit for any of the git’s args.

1. describe

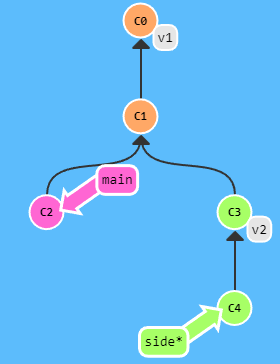
git describe <optional ref>

This arg returns a string of this format

<tag>\_<num of commits>\_g<commit hash>

git describe returns a string that defines how many commits there are between the given commit or if not specified, the HEAD, and the closest parent commit with a tag.

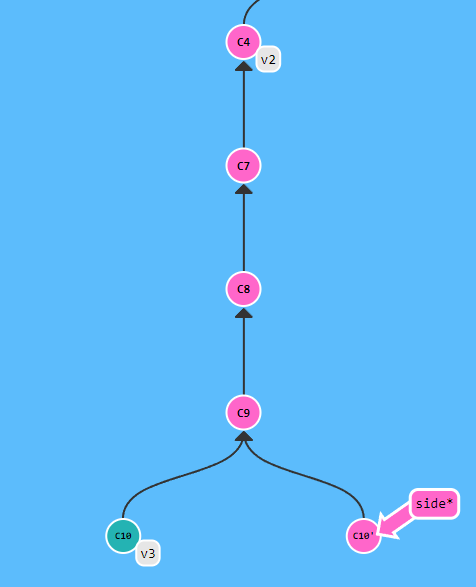
For ex.:



git describe main would return

v1\_2\_gC2

Another ex.:



git describe side returns v2\_4\_gC10’ and not v3 because C10 is not an ancestor of C10’.

1. Remote Repository

A remote repo is nothing but a replica or clone of the local repository with its own commit history which is synced with the local commit history.

This allows collaboration and also provides a backup or central location for any project/repository. Popular online remote repo platforms include Azure Repo, Github, Gitlab etc. We can make our own too but these are free and provide all important features with stability.

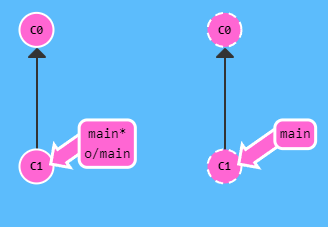
Working with remotes is simply transferring commit states between local and remote repo.

1. clone

If we already have a remote repository set up, then git clone clones the remote repository into our local directory along with the commit history.

git clone <.git url>

For ex.:



git clone of the repo containing the commit history on the right will create the local repo on the left.

Here o/main stands for origin/main, origin is the default name for any remote repo and here it means o/main is the remote repository’s ‘main’ branch. The format is <remote name>/<branch name>.

When we attach a local repo to any remote repo, the local repo gets 2 branches for each branch in the remote, first is the local branch (main) and second is the remote’s local branch (o/main) called remote branch in short. The local branch can be moved around however.

A remote branch has special properties, firstly it can’t be checked out (puts HEAD in detached mode on checkout) so it can’t be manually mutated. Secondly, it represents the state of the remote repository’s same named branch (as of last sync, local repo will not update with the current state until manually synced with the remote repo if remote repo got updated).

1. fetch

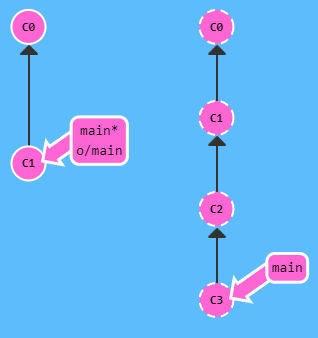
This arg fetches the current state of the remote repository and updates the remote branches.

git fetch

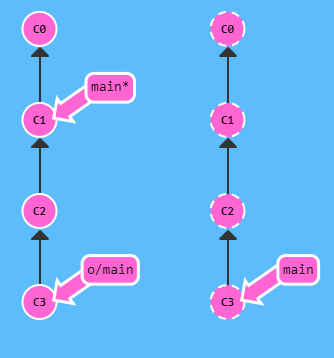
It does mainly 2 tasks, firstly downloads all commits not in local repository but in remote repository and updates all remote branches.

git fetch doesn’t modify any local branch and since we can’t checkout any remote branch, even if it updates the remote branch, our checked out state or local repo state is never affected either.

For ex.:



git fetch here will do so



It simply downloads the commits and moves the remote branch.

* Flags

1. colon refspec

Just like git push, git fetch can specify the remote branch and the local branch when fetching commits.

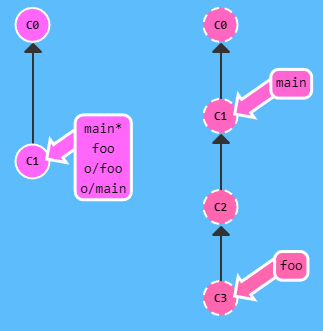
git fetch <remote repo> <branchName>

//or

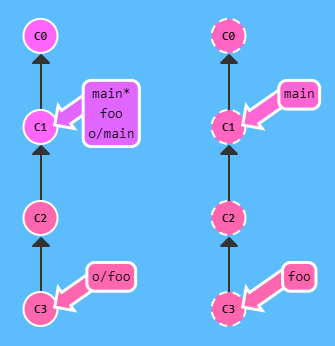
git fetch <remote repo> <Remote source>:<local source>

the colon refspec is flipped here in relation to git push’s colon refspec.

For ex.:



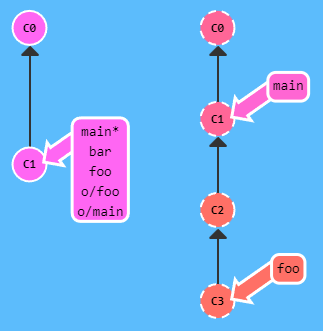
git fetch origin foo, will do



It says fetch foo’s status on remote origin and update the remote branch tracking it.

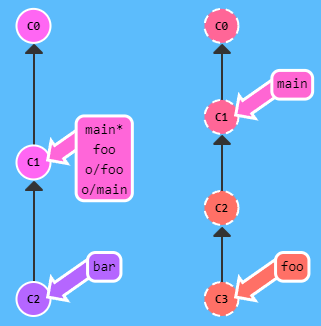
For the colon refspec variant, it can fetch any commit or branch and update the local source branch provided to it.

For ex.:



git fetch origin foo^:bar

will do



It says, fetch all the commits not in local repo until C2 and update the branch bar. This is a bad practice as fetch is intended to not update the local branches, only the remote branches. Furthermore, only branches not checked out can be fetched to.

If the local branch to be fetched into doesn’t exist then it is created where HEAD is at and then the fetch is ran.

If in the second variant, the source branch doesn’t exist, it is created at the same place as the remote source branch.

Empty remote source in git fetch creates a new branch at HEAD

so

git fetch origin :x

creates a new branch x at HEAD.

1. pull

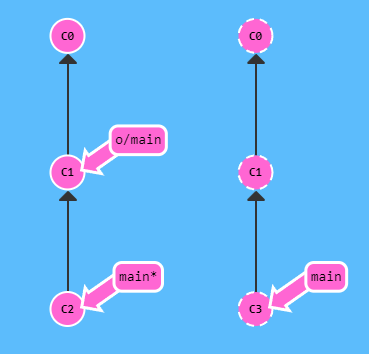
After fetching, we can incorporate the changes from the remote branch into our local branch using args like git merge o/main, git rebase o/main, git cherry-pick o/main etc. But git provides an arg to simplify that.

After fetching, if the remote branch is on a different commit than the local branch then we can use git pull (we can use it otherwise too, it doesn’t do anything). This either merges remote branch’s commit into our local branch commit or rebases our current local branch commit on top of the remote branch commit.

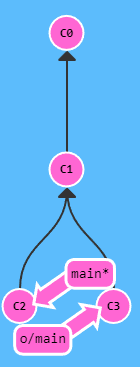
git pull

By default, it merges the latest commit.

For ex.:

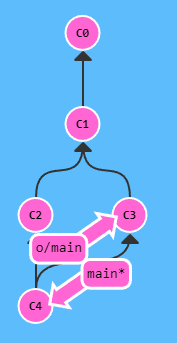


If our local repo (on left) and remote repo (right) are like so, then git fetch will do so



and then we can merge this new commit into our local branch using

git merge o/main



Or we can get the exact same result, if we just do git pull. Some versions of git don’t include git fetch in a git pull, such as Visual Studio’s Git, but normal git from git-scm does.

* Flags

1. --rebase

If we want git pull to do git rebase instead of git merge, then we specify this option.

1. Colon Refspec

Just like git push and git fetch, we can specify remote and then specify where to pull from and merge/rebase to.

git pull <remote repo> <branchname>

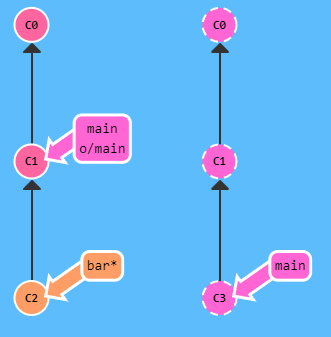
//or

git pull <remote repo> <source>:<destination>

The first variant fetches the given branch from the remote and updates the remote branch tracking it, then merges the remote branch into the checked out branch.

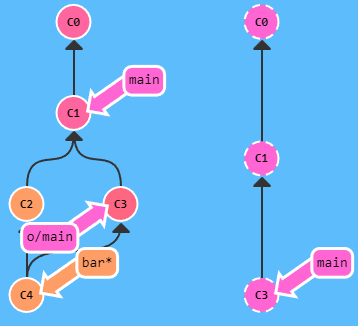
The second variant fetches the destination branch on remote repo and then updates the source branch, then it merges the source branch into the checked out branch.

For ex.:



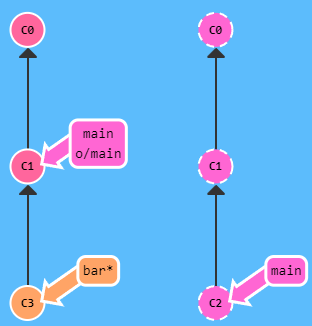
git pull origin main

makes



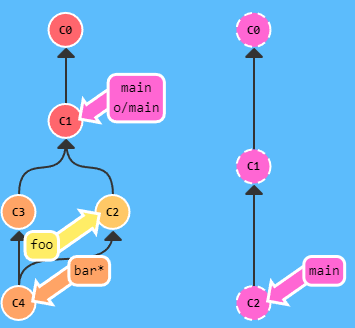
It fetches remote’s main branch and updates the remote branch tracking it, i.e., o/main, then it merges the remote branch into the checked out branch.

Another ex.:



git pull origin main:foo

makes

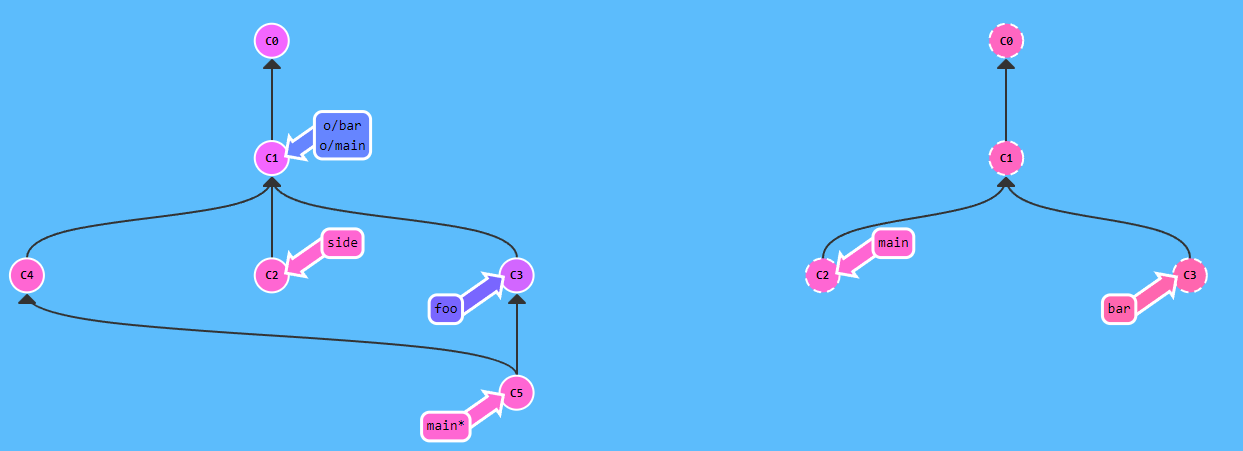


There is a third variant of git pull that uses the same format as git fetch

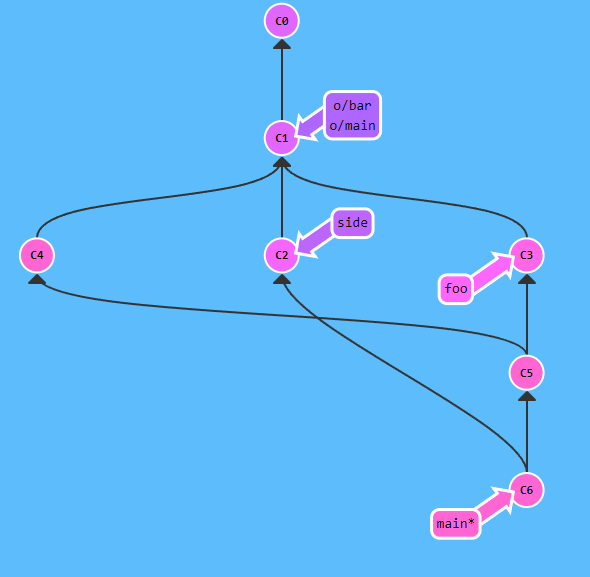
git pull <remote repo> <remote source>:<source>

This variant works like the second variant, but it can also create a new source branch if it doesn’t exist in the same place as the remote source, exactly like git fetch.

For ex.:



In this situation, if we do git pull origin main:side, it takes main on remote and updates side on local. So it runs git fetch main:side, in this case C2 is already the same as remote’s C2 so it doesn’t do anything and then merges side into main.



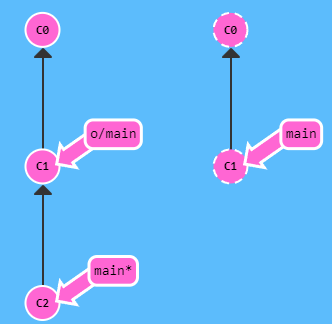
1. push

Git push pushes commits in local repo (which are an ancestor of a branch and not dangling commits) not in remote repo to the remote repo and updates the remote’s branches to be at the same commit as the local’s commits, then it also updates the remote branches in the local repo to be at the same commit as well, thereby updating both remote and local’s remote.

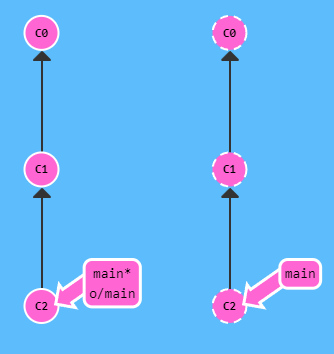
git push

Without any options, git push’s behavior depends on push.default file.

For ex.:

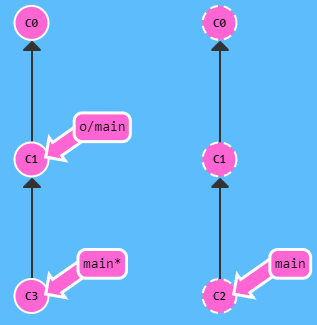


git push here would make it like so



* We can’t push if there are latest commits (branches) in the remote that aren’t merged/rebased into any of the child or parent commits of the local repo (of the same branch).

For ex.:

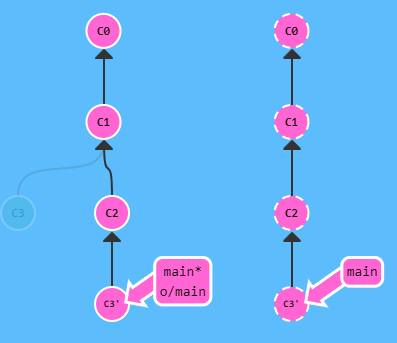


git push would fail here

If we do git fetch, then do git push it would still fail as we haven’t incorporated the remote’s changes yet.

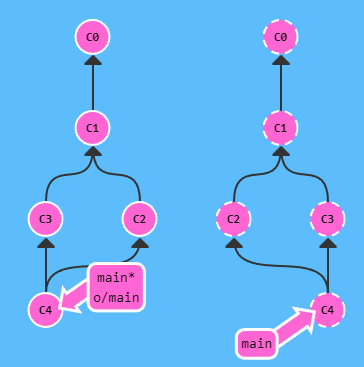
If we do git pull, or git fetch then git merge/rebase then do git push, that would work.

Like so



Here we did git rebase, so C3 is a commit which is not an ancestor of any branch/ branch being pushed, so C3 isn’t pushed or rather it is optimized away.

However, if we do git merge then both C3 and C2 would be an ancestor and so both would be pushed



* Flags

1. Remote

We can specify where we wish to push and what branch to push.

git push <remote repo> <place>

//or

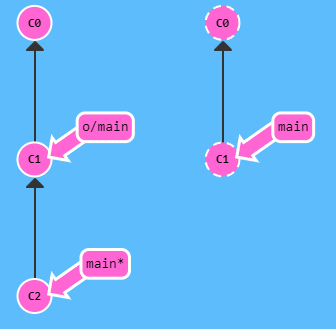
git push <remote repo> <source>:<destination>

For ex.:

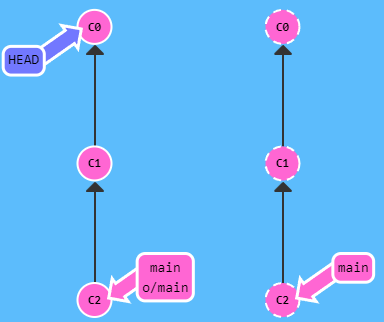
git push origin main

means push local’s main branch onto main branch in remote repo named origin.

Another ex.:

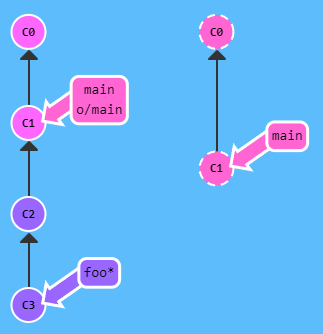


Here even if we git checkout C0, then do git push origin main, then we will get



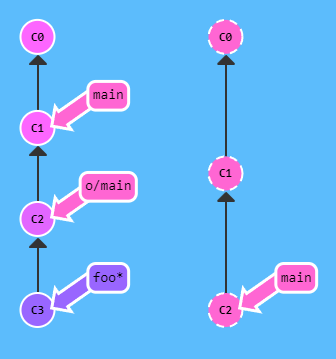
The second variant uses colon refspec, it is a way to specify different branches on local and remote.

For ex.:



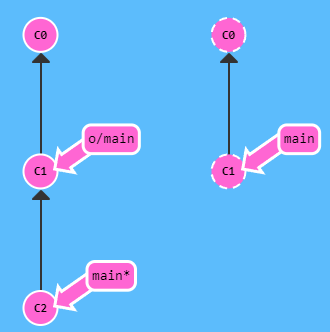
Here if we do git push origin foo^:main

then we get



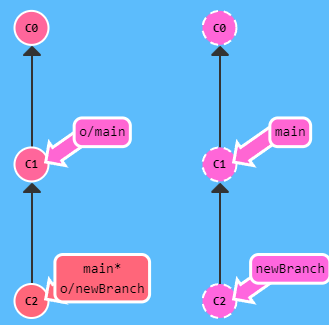
as it is saying push C2 to remote’s main branch where the remote is named origin.

Another ex.:



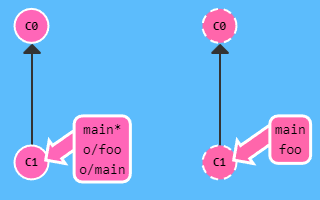
If the target remote branch to push to doesn’t exist then it is created.

So if we do git push origin main:newBranch then we get



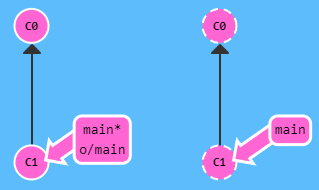
If we omit source, then we push “nothing” into a given branch onto remote repo, thereby deleting it.

For ex.:



git push origin :foo

makes



1. Pull Request

If the remote has disabled direct pushes for certain users or globally then we can get an error like so



It is both a good practice and enforced in this case to use Git Pull Requests.

PR is a request to pull given commits on the remote repo into another branch in the remote repo. So we need to first push the changes to another branch on the remote and then create a PR to any other branch, then the commits to be applied/pulled are checked for “merge conflicts” and then if there aren’t any and upon approval/any other processes in the remote repo, the commits are finally pulled into a remote branch and the remote repo and it’s branch is updated in the remote with the commits. Finally, the PR is closed.

To create a pull request, we can either do it through the remote repo platform or use

git request-pull <start commit> <url> <end commit>

url is of the repo we are opening a PR to. start and end commit can be commits, or tags, or even branch names. If we omit end commit then HEAD is taken. If we use branch names/HEAD then the PR gets mutated with the new commits if branch names/HEAD points to a newer commit.

For ex.:

git request-pull v1.0 https://git.ko.xz/project master:for-linus

says create a PR from tag v1.0 on given URL to the local’s master branch which is called for-linus in the remote repo.

1. Remote Tracking

Each branch on local that is being synced with a branch on remote repo is tracking it, this tracking property can be modified as well. This tracking property sets the merge target for git pull and push destination for git push.

By default, if none is specified then if we clone a repo, the remote branches are named like so <remote name>/<branch name> and this same <branch name> is assigned to the local branch and local branch is set to track this remote branch.

There are 2 ways to change the remote tracked branch,

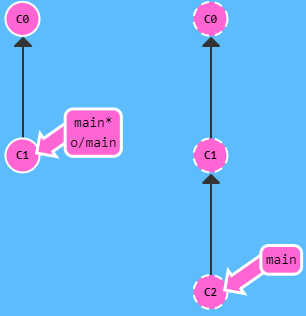
git checkout <branchname> <remote branch>

or

git branch -u <remote branch> <branchname>

The only difference is that this doesn’t checkout the given branch.

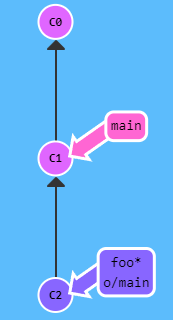
For ex.:



git checkout -b foo o/main

and then git pull

would do



and remote repo remains the same.

Similarly, if foo tracking o/main pushes any commit, it updates the main branch in the remote repo.

1. Setup

To setup git for a local repo.

1. init

git init

will initialize the current directory to be used/tracked with git and add a .git folder in it.

1. config

Before making any changes (commits/pushes/etc.) we need to tell git the user identity. We do so with the config arg. If we add a --global arg to the config then the config will be applied to git globally and not just for current git directory.

It takes a key value pair.

git config --global <key> <value>

2 keys are necessary,

git config --global user.name “myusername”

git config --global user.email “myemail@yoo.com”

1. Optional branch

We need to specify which branch we are going to be working on,

git branch -M main

-M is not required but it forcibly moves the current changes to the new branch, or rather renames the current branch to “main”.

This is optional because by default git is already on “main” branch.

1. add

git add <file/folder 1> <file/folder 2> …

//or

git add .

will add the given files/folders or all files and folders to be staged for the next commit. Staging means these files will be commited.

If there is a .gitignore file in the root of the current directory, then all files/folders in it will be ignored by git add and will never be staged.

* 1. .gitignore

This is a special text file that tells git to never stage the files/folders given in it.

Each line in it gives a pattern, then when git add has to add a file it checks if the pattern exists in .gitignore, if true then the given file is ignored (Simplified, in reality this matching is likely more complex).

The pattern for gitignore is given in the manpage for gitignore

<https://git-scm.com/docs/gitignore>

Sample .gitignore file

# Java class files

Android/\*\*/.class

Ignores all files with name “.class” in any subfolder inside Android directory in the current directory.

So if we have <Current Dir>/Android/yo/yooo/yoo/.class or <Current Dir>/Android/.class

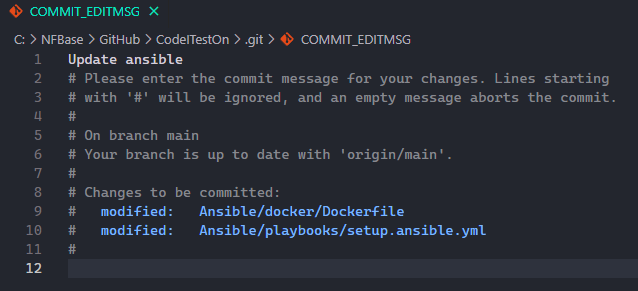
then both these files will be ignored.

1. commit

git commit

will add all staged files to a commit, git will also present an interface to write the message for the commit as messages are mandatory.

It may look like so



Then, closing the editor will bring us back to the cli and the commit will be created.

1. Remote Repository

Now we need a remote repository where the commit will be synced to, to create this repository we can use popular repository hosting platforms like Github or Gitlab and the process to create the repo varies with each platform.

For github, we can create the remote repo using the website or using “gh” which is the github cli tool.

Then after creating the remote repository we just need its .git URL,

It looks like so <https://github.com/cryoelite/T1.git>

Next we need to add this remote repo to our current git directory, to do so

git remote add origin https://github.com/cryoelite/T1.git

Here remote is the main arg, add adds a new remote repo, origin is the shorthand name of the remote repository.

1. push

git push -u origin <branch>

<branch> is generally main. Pushes current branch to main on remote repository called “origin”.

Here -u flag creates a link between current branch and the given branch on remote after pushing to it.

Git will present an interface to either login to the remote repository or specify an API token. Specifying either will allow git to push the changes.