

**Introduction:**

In this section we will talk about one of the most important and powerful features of GIT called *branching*.

**We will learn how to**:

🡪 Use branches to diverge from the main line of development and work on something else in isolation.

🡪 Compare branches to see their differences.

🡪 Merge branches (*different merging techniques like 3 way merging, fast forwarding, squash merging and rebasing*).

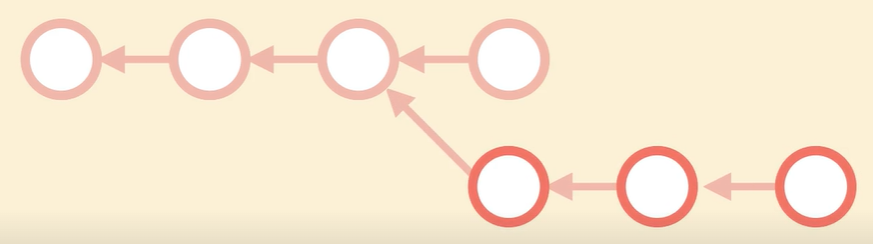
🡪 Resolve merge conflicts.

🡪 Undo a faulty merge.

🡪 Essential tools (*stashing, cherry picking*)

**What are Branches**:

Well branching allows us to diverge from main line of work and work on something else in isolation.



Conceptually, we can think of a branch like a *separate isolated workspace*. So we have our main workspace which is called *master*.

Now we can have another workspace called *feature* for working on a new feature in isolation.



*While we are developing this new feature, our code might become unstable, so we don’t want to release the code in this workspace*.

So *we continue working here and when we are done, we test our code and after we fix all the bugs then we bring the changes in master workspace*. This is called *merging*.

Branching allows us to work on different work items without messing up with the main line of work. We keep the main line as stable as possible, so we can release it anytime.

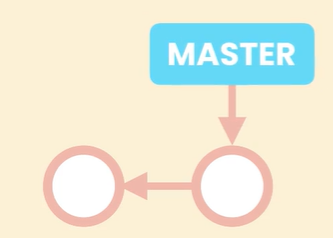
But…



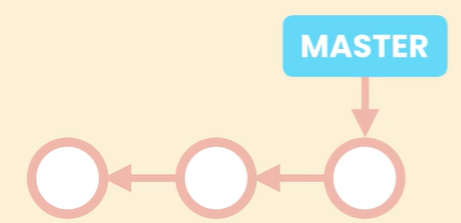
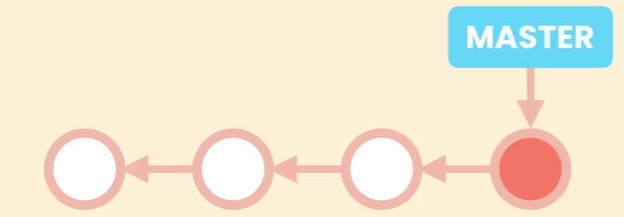
The way GIT manages branches is very different from many other version control subsystems like *subversion*.

Note: In subversion when we create a new branch, subversion takes a copy of our entire working directory and stores it somewhere else. If we have 100s or 1000s of files, all these have to be copied and this operation will take a while.

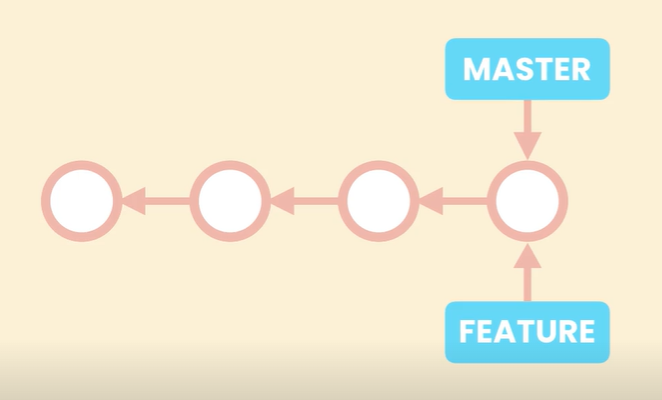
On the other hand, GIT branches are superfast and cheap because *a branch in GIT is just a pointer to a commit*.

 So the *master* branch is just a pointer to the last commit in the main line of work.

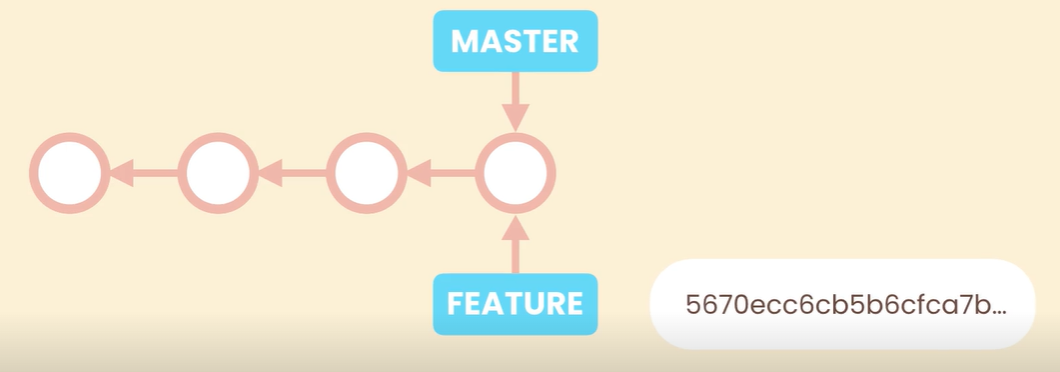
As we make new commits GIT moves this pointer forward automatically, so it knows what is the latest code in main line of work (*that’s the snapshot stored in this commit*).

🡪🡪 

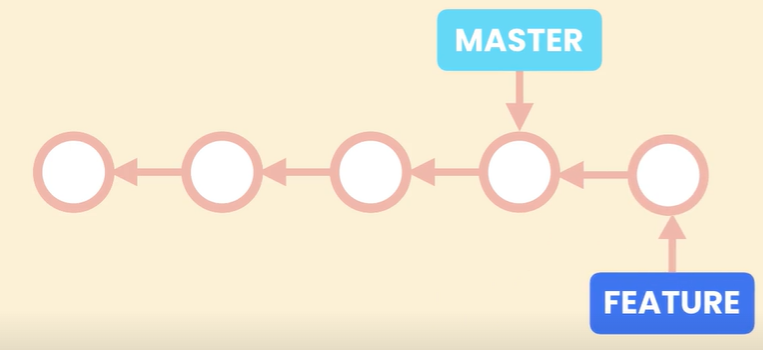
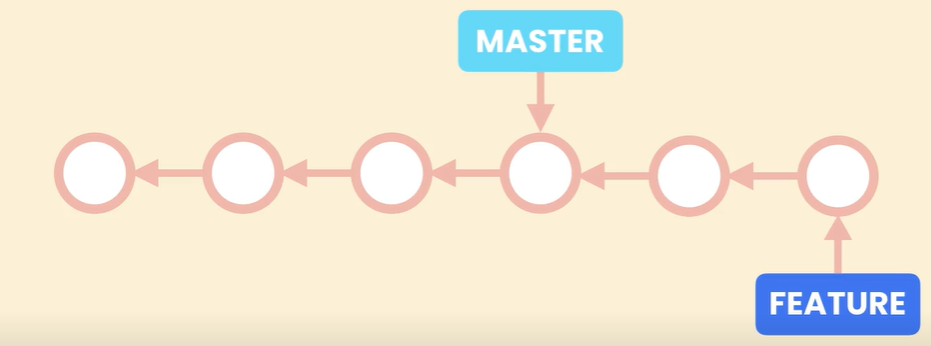
*When we create a new branch*(*feature*), *GIT creates a new pointer that can be moved around*.



*This pointer is just a tiny file that contains a 40byte commit ID, that’s why creating a branch in GIT is blazingly fast*.

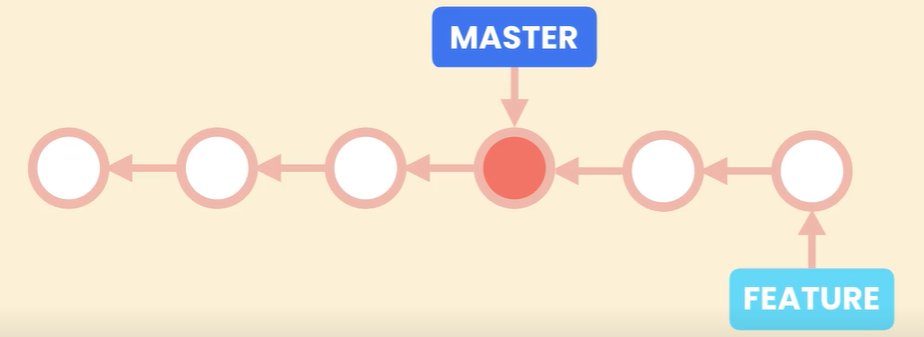


When we switch to this branch and make new commits GIT moves this pointer forward.

🡪🡪 

The master pointer stays where it is, so GIT knows the latest code in each branch.

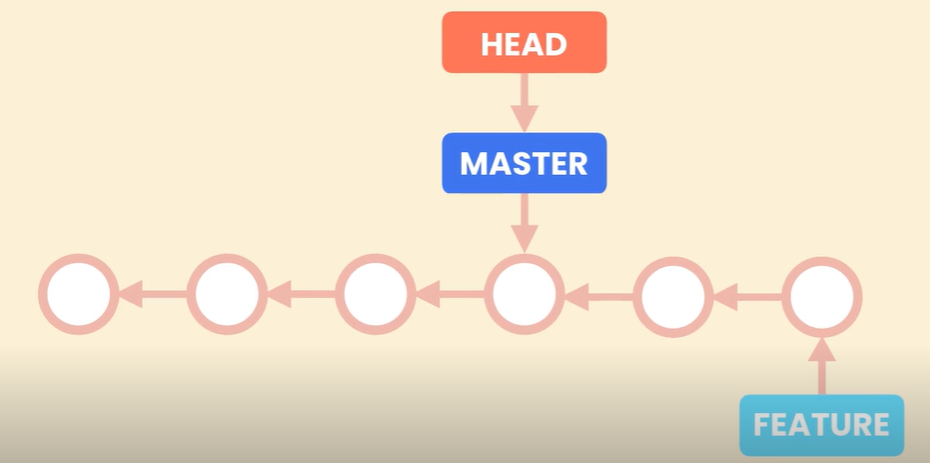
When we switch back to master, GIT takes the snapshot from the commit that master points to and resets our working directory to that snapshot.



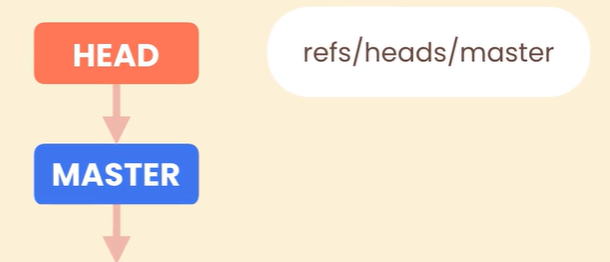
So we always have a single working directory.

*Now how does GIT know which branch we are currently working on*?

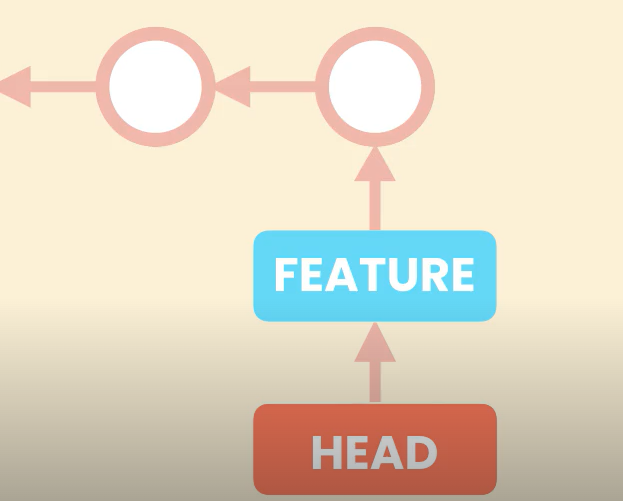
By using a simple pointer called ***HEAD***.



This pointer is also another tiny file that contains the name of a branch like master.



Let’s say we switch to another branch, GIT moves the head pointer out. So it updates a tiny file and writes the name of target branch.

(*refs/heads/feature*)

That’s how we can track which branch we are currently working on.

Throughout this section we are going to learn everything we need to know to work with branches.

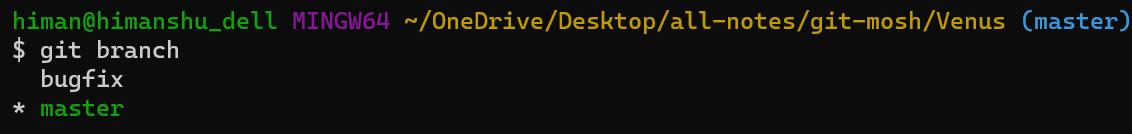
**Working with Branches**:

Let’s say we just got a bug report, now to fix this bug first we should create a new branch called *bugfix*.

git branch bugfix

To see the list of branches we have, we type,

git branch

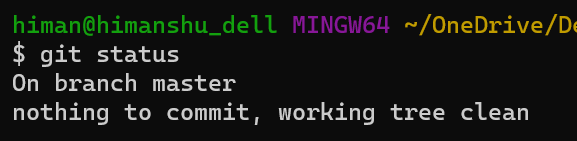


Currently we have two branches bugfix and master.

There is an \* asterisk before master, which means we are currently on the master branch.

There is another way to know current branch.

git status



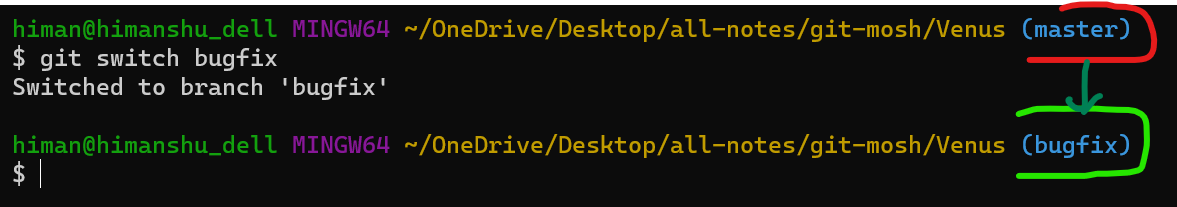
Enough of detour, Now to fix this imaginary bug, first we need to change the current branch to *bugfix*.

There are two ways to do this,

🡪 In the past, we used checkout command. But this command has multiple applications and a lot of people find it confusing.

🡪 These days we have a new *command for switching branches*.

git switch bugfix

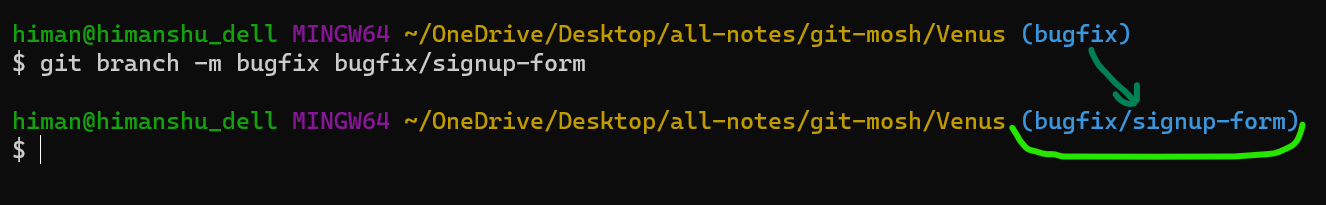


Now we are on the bugfix branch. But before going forward, if we want to rename this branch(*for specifying which type of bug we are trying to fix, a recommended strategy*).

git branch -**m** <old-name> <new-name>

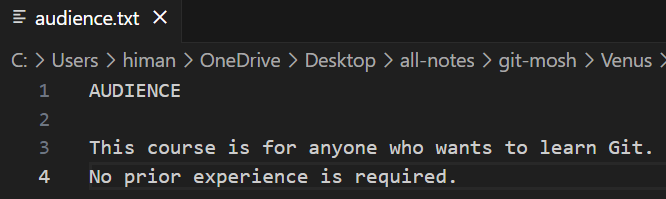
First we type old name after -m and then new name.

To name a branch we can use a convention like *bugfix-signup-form* or *bugfix/signup-form*(*assuming that this bug is in our signup form*).

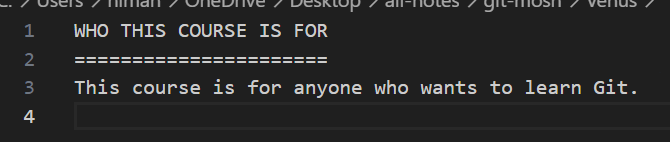


So to make some changes let’s open audience.txt file.

code audience.txt (*it will open the file in VS code*)

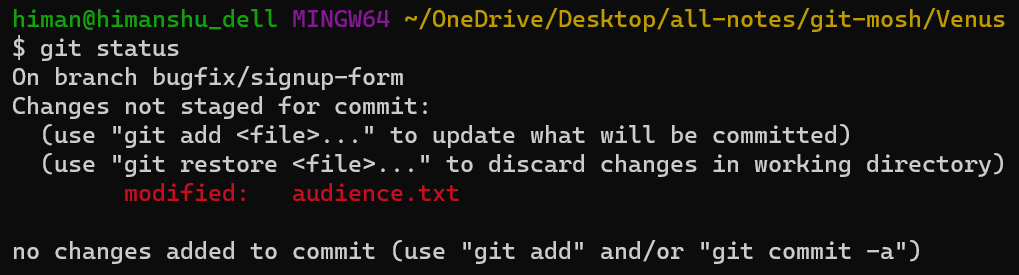


After changes,



Here we did 3 kind of changes *addition*, *modification* and *deletion*.

If we run git status, we can see audience.txt is modified and this file is not staged.

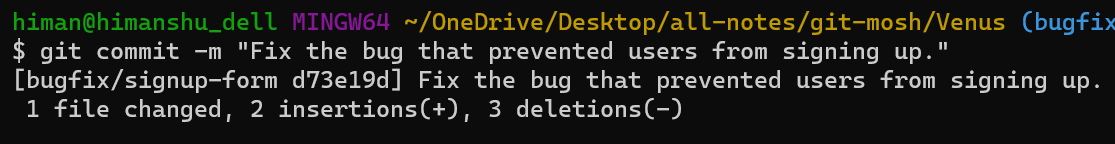


Let’s add it to staging area,

git add .

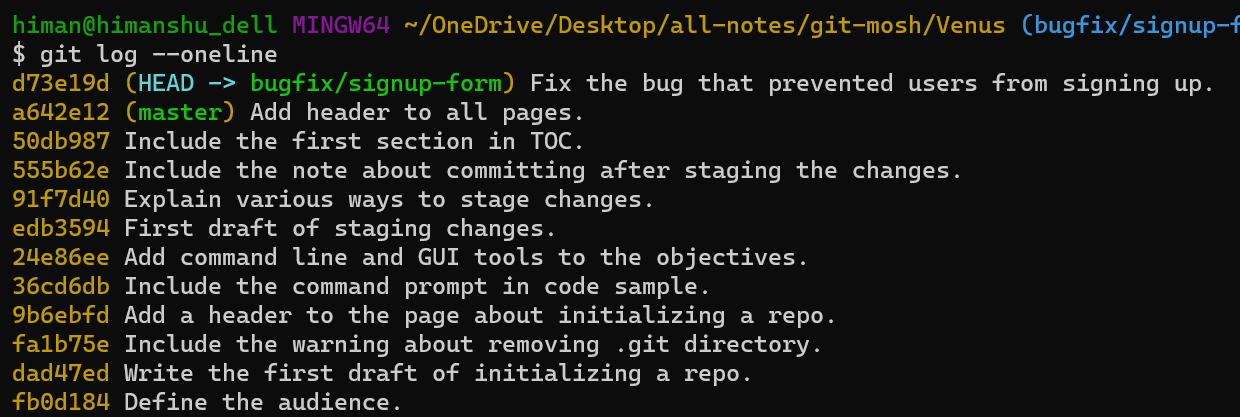
And then commit it,

git commit -m "Fix the bug that prevented users from signing up."



Now, let’s look at our log,

git log --oneline



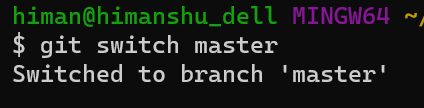
On the top we have HEAD 🡪 bugfix/signup-form (*HEAD pointing to bugfix/signup-form branch*).

That means currently we are on the bugfix branch and d73e19d is the latest commit on this branch.

Below that we can see our *master* branch which is just a pointer to a commit.

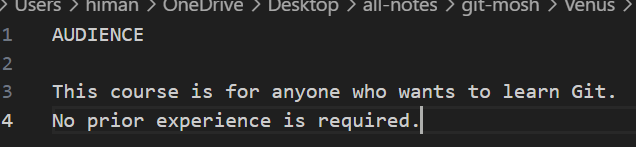
Here we can say *our master branch is one commit behind the bugfix branch*. At some point we are going to merge the bugfix branch with master branch to bring the master branch up.

So in the audience.txt we made a few changes and *these changes are only visible in the bugfix branch*. If we switch back to master branch.



And open audience.txt.

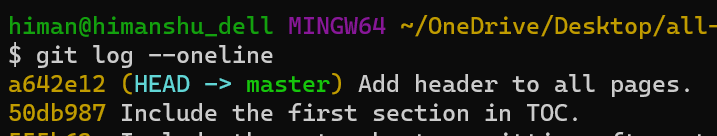
code audience.txt



We see old version of this file.

***Takeaway***: Branches allows us to work on different tasks in isolation, the code we have in these branches are isolated and at some point we merge them to bring them together.

While we are on master branch, git log –oneline

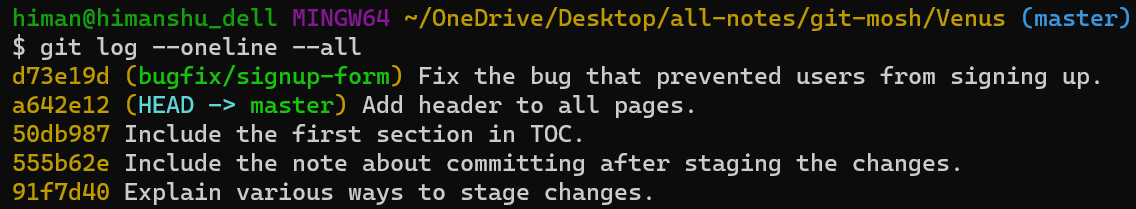


On the top HEAD is pointing to master branch, but we don’t see the bugfix branch.

It is because *bugfix/signup-form* branch is ahead of the master branch. So its commits are not listed here by default.

To *view the commit across all branches*, use all options.

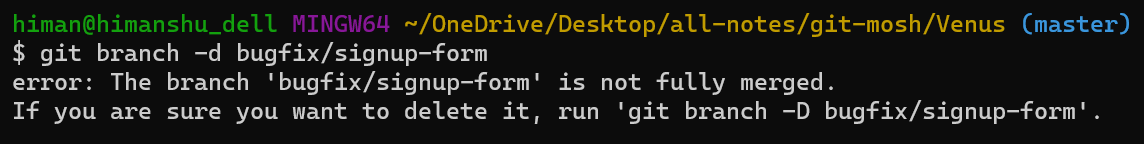
git log --oneline --all



Now even though we are on the master branch, we can see the commits in other branches.

Sometime in the future when we are done with the bugfix branch (*when we merge it into the master branch*), we need to delete it.

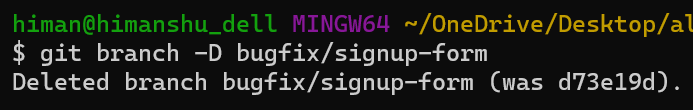
git branch -d bugfix/signup-form



Here we got an error saying the branch is not fully merged. *By default GIT prevents us from accidently deleting a branch unless we merge it first*.

But if we are sure we don’t want changes in this branch, we can force the deletion by using -D option.

git branch -D bugfix/signup-form



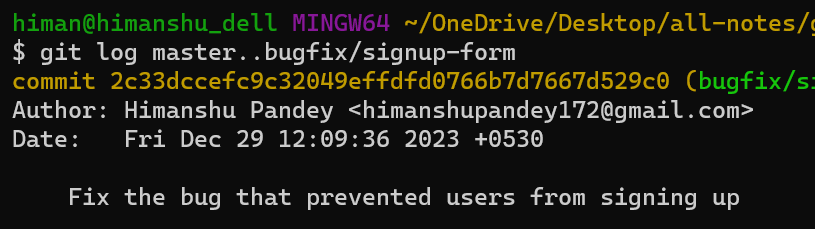
**Comparing Branches**:

As we commit to our branches, we need to know how they are diverging from master. So in this lesson we will learn a couple of ways to compare branches.

Currently we are at master and at some point in the future we are going to merge the bugfix branch into master. But before doing so *we want to know what commits are coming in master*.

git log master..bugfix/signup-form

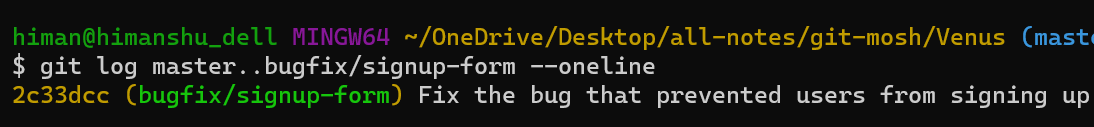
This means show me *all the changes that are in bugfix branch but not in master branch*.



Currently we only have a single commit, if we had more commits all of them would be listed here.

Once again we can use --oneline option to make the output more concise,

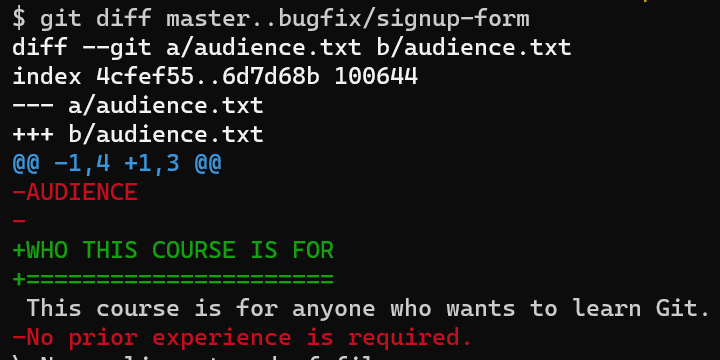
git log master..bugfix/signup-form --oneline



What *if we want to* ***see the actual changes*** *and not the list of commits*?

We use diff tool to compare master with bugfix branch.

git diff master..bugfix/signup-form

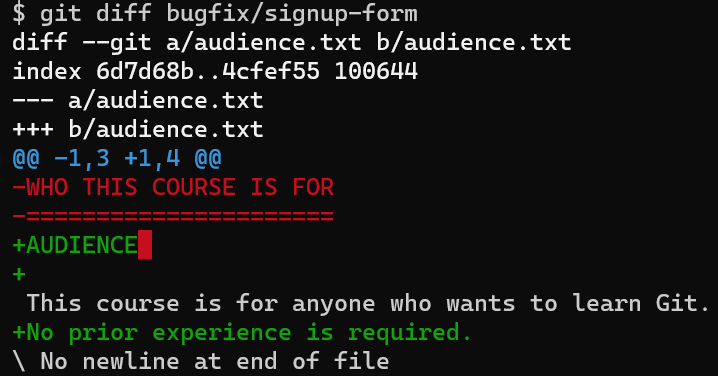


So we know that once we merge bugfix branch with master, the audience.txt is going to have these changes.

There is a shorter way to write this command, since we are in master we don’t have to type *master..* in this diff command.

We can straight up write,

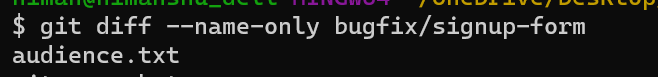
git diff bugfix/signup-form

🡨same result.

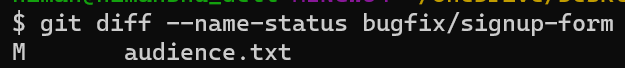
Sometime we don’t want to see the changes in the code, we just *want to see which files are affected*.

We can use --name-only OR --name-status options.

git diff --name-only bugfix/signup-form



git diff --name-status bugfix/signup-form



This means when we merge bugfix branch with master, the audience.txt (*M*) file is going to modified.

**Stashing**:

Earlier we talked about when we switch branches, *GIT resets our working directory to the snapshot stored in the last commit of the target branch*.

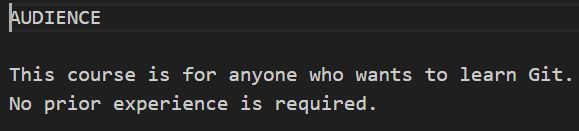
*If we have local changes in our working directory that we have not committed yet, these changes could get lost* in these situations and git does not allows us to switch branches.

Let us see how this happens and what can we do about it?

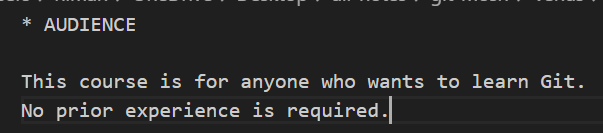
In previous lesson we made some changes in audience.txt in our bugfix branch.

Now currently we are on master branch, so let’s open this file,

code audience.txt



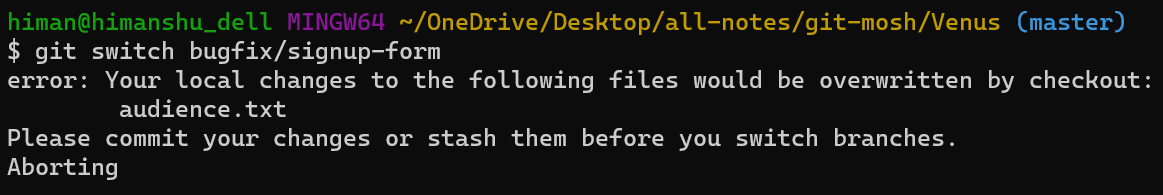
And make some other kind of changes here…



Here we just added an asterisk to indicate a change.

Now let’s say we need to quickly switch to bugfix branch,

git switch bugfix/signup-form



We get an error saying local changes to the following files would be overwritten by checkout. Please commit your changes or stash them before you switch branches.

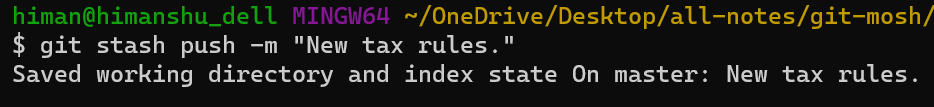
In this case we don’t want to commit these changes yet. So in situations like these *we can* ***stash*** *our changes*.

**“***Stashing something means storing this in a safe place, so we are going to store this somewhere in our GIT repository, but this is not going to be part of our history***”**

We do it this way,

git stash push -m "New tax rules."

With this message we can tell what these changes were all about. (*Let’s say we are working with new tax rules then we quickly needed to switch to bugfix branch*).



Note: By default new untracked files are not included in your stash. Say for example let’s add a new file in our working directory.



git status -s

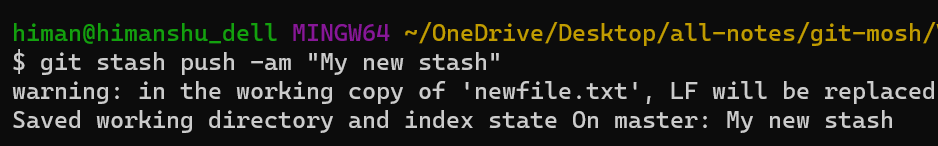
 🡨 untracked file.

If we stash the changes this new untracked file is not going to be included in the stash by default. *To include an untracked file into stash we need to use all option*.

git stash push -**a** -m "My new stash"

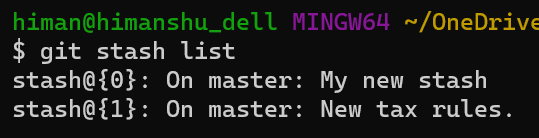
or

git stash push -**a**m "My new stash"



*To find out all the stashes we have*,

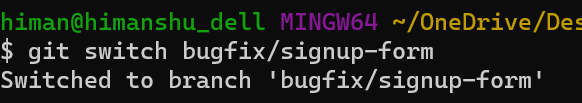
git stash list



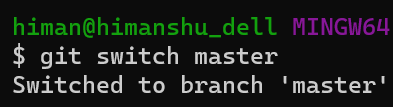
So we have two stashes and each stash has a unique identifier (*like stash@{0} or stash@{1}*). Here 0 and 1 are index.

After creating our stash, our working directory is clean, so we can switch to bugfix branch now.

git switch bugfix/signup-form



Now we can do our work here and when we are done , we can switch back to master.

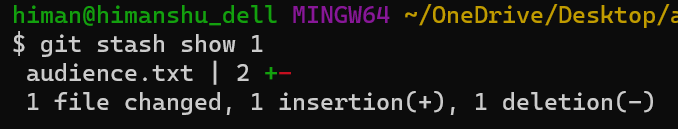


At this point we want to apply the changes from one of our stashes to our working directory. But before doing that we might *need to look at changes like which lines of code have been modified*.

git stash show stash@{1}

OR

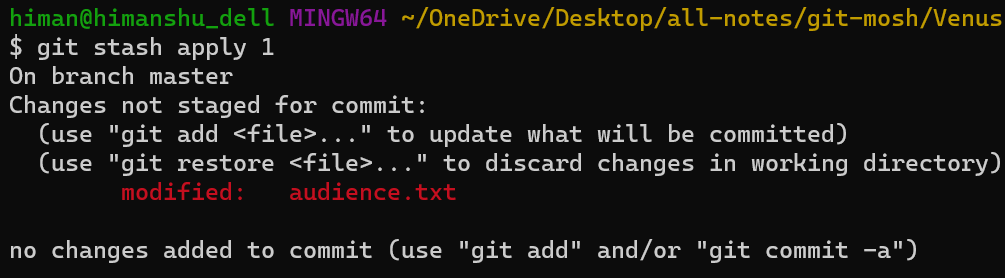
git stash show 1



Now we know that in stash@{1}, a file has been changed with 2 changes 1 insertion and 1 deletion.

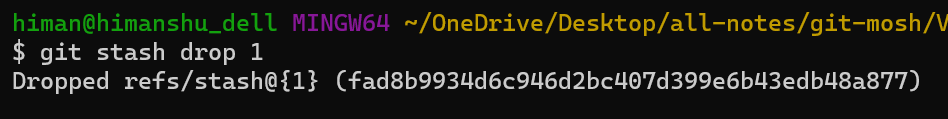
So we can go ahead and *apply this stash in our working directory*.

git stash apply 1



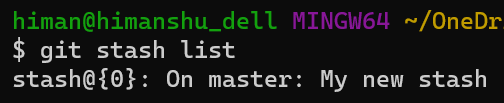
The changes are applied to our working directory. Now when we are done with this stash, we need to *remove this stash to clean things up*.

git stash drop 1



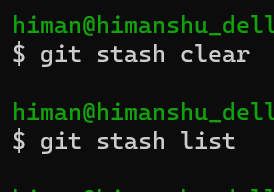
If we look at our stash list again,

git stash list



To *remove all stashes*,

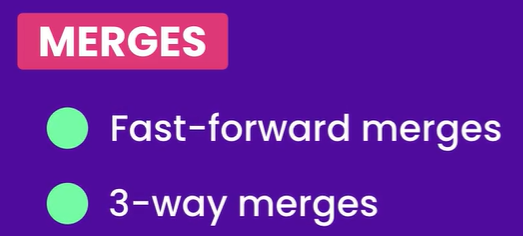
git stash clear



All our stashes are gone now.

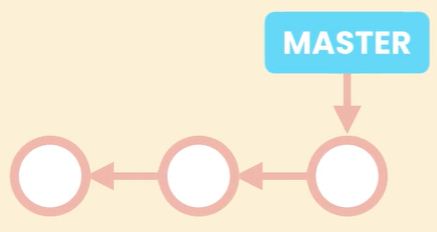
**Merging**:

Merging is all about bringing changes from one branch to another. In GIT we have two types of merges.

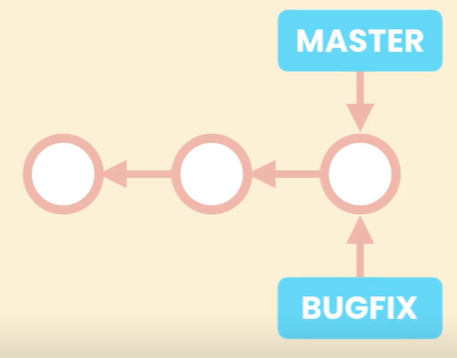


Let’s study each of these merges using a real example.

Here is the master branch with 3 commits.

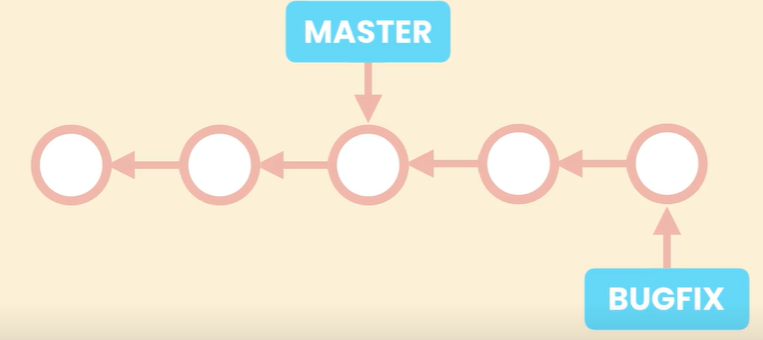


We create a new called bugfix. (*in git a branch is just a pointer to a commit*).

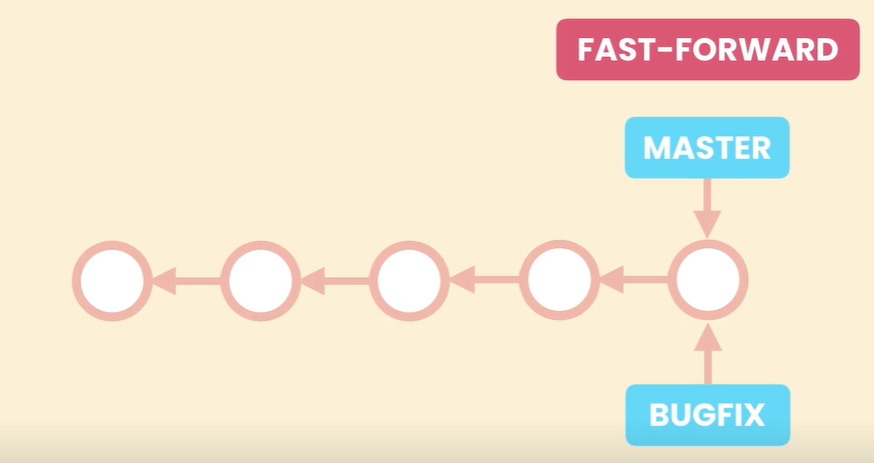


In this case both master and bugfix pointer are pointing to the same commit.

Then we move to bugfix branch and make a couple of commits and when we are done here we need to bring the changes back to master.

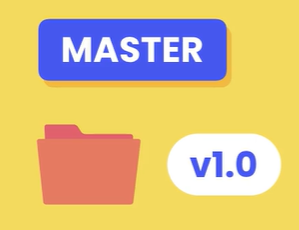


*Since these branches have not diverged and there is a direct linear path from bugfix to master, all GIT has to do to merge the changes is to bring the master pointer forward*. This is what we call a *fast forward merge*.



If this is confusing, let’s understand it from a simple example.

Forget about GIT for a while and imagine we have a directory called master and code in this directory is at version 1.0.



We take a copy of this directory and call it bugfix.



At this point in time, the code in these directories is identical.

Now we start making some changes to the bugfix directory. So we get to version 1.2.

 🡪 🡪 🡪

We are done here and now we want to bring the changes back to master.

Now how do we do this?

One way is to copy all the files from bugfix into master. But this is not the best way to solve this problem. What if we have thousands of files.

There is a better way.

*Since we don’t have any additional changes in master and essentially what we have in master is the first version of bugfix. We can simply rename bugfix to master*.



So from this point forward, we can say this is our new master directory. The idea of fast-forward merge in GIT is very similar.

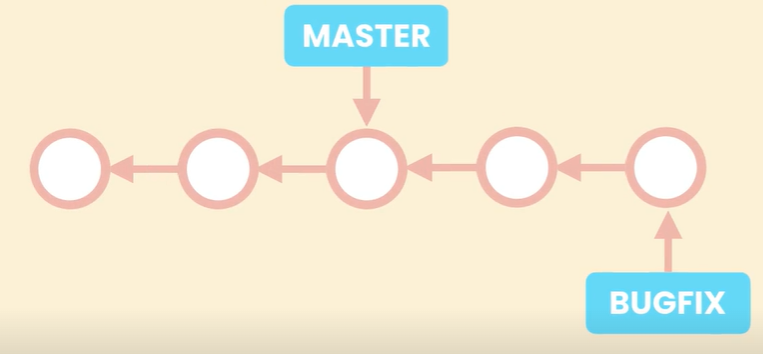
So If two branches have not diverged and there is a *direct linear path from the target branch to the source branch*, git runs a fast forward merge.

*It simply brings the pointer of the source branch forward*.

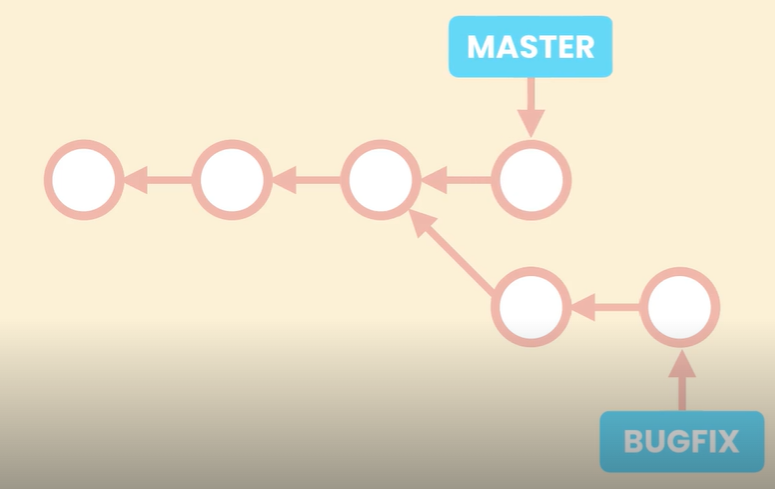
Note: Once we are done with the bugfix branch we can simply remove it, which removes the pointer.

Now Let’s look at an example of *3 – way merge*.

In this example, bugfix branch is 2 commits ahead of the master.

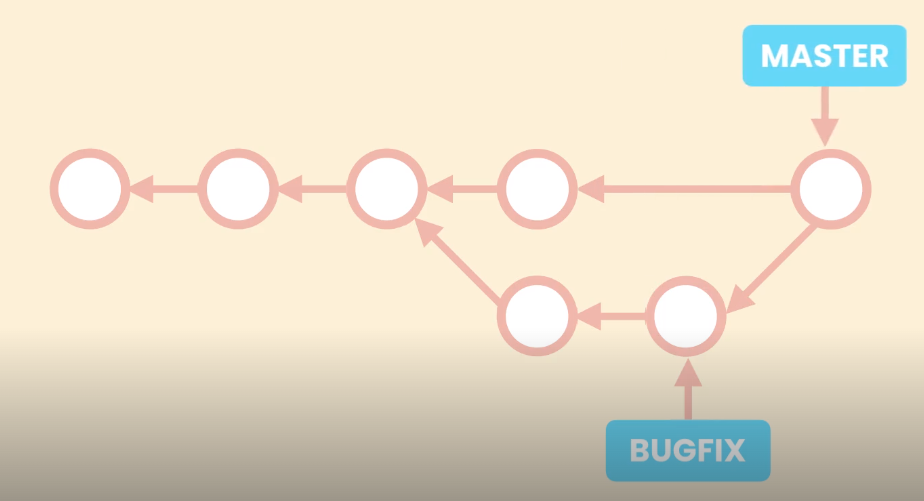


But before we merge it with master, we go back to master and add an additional commit.



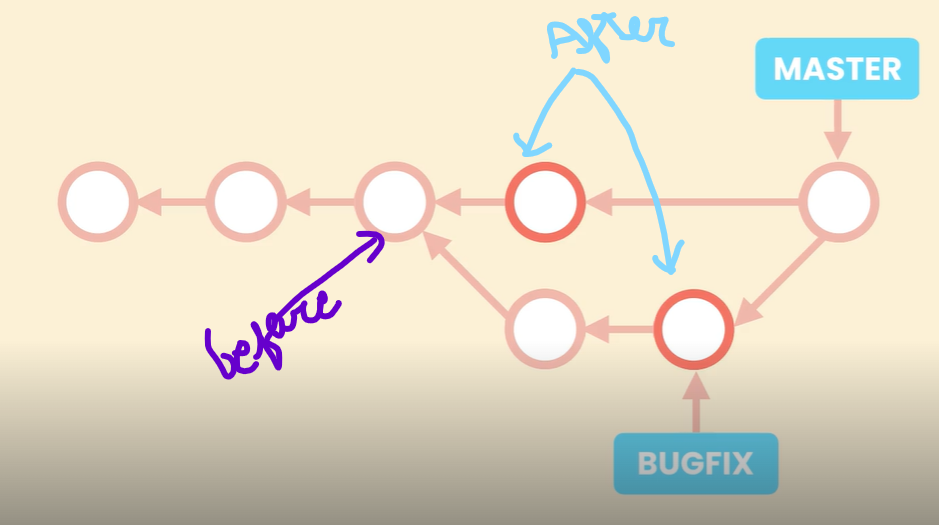
*So we have some changes in master that don’t exist in the bugfix branch. So if we run a merge, GIT cannot move the master pointer forward and have it point at the same commit as bugfix, because otherwise we will loose the latest commit in the master branch*.

Therefore *when we run a merge GIT creates a new commit that combines the changes from these two branches.*

**

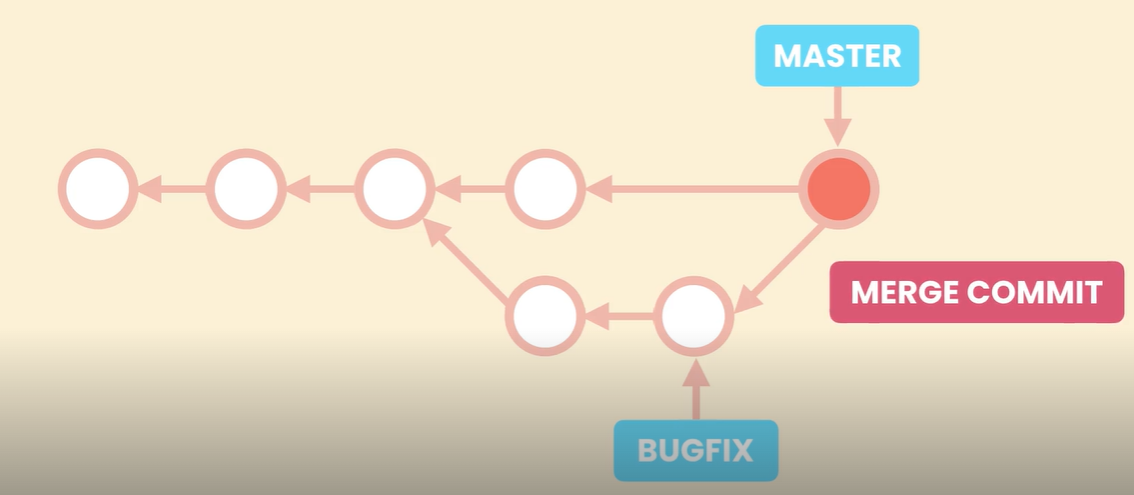
*The reason it is called a 3 – way merge is because this new commit is based on 3 different commits*.

The common ancestor of our branches which includes the before code and the tips of our branches which contains the after code.

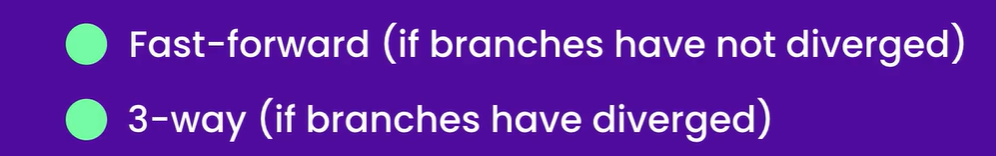


So GIT looks at 3 different snapshots (*one before snapshot and 2 after snapshots*). Based on these it will figure out how it should combine the changes.

Hence we create this new commit which is called a *merge commit*.



Recap:



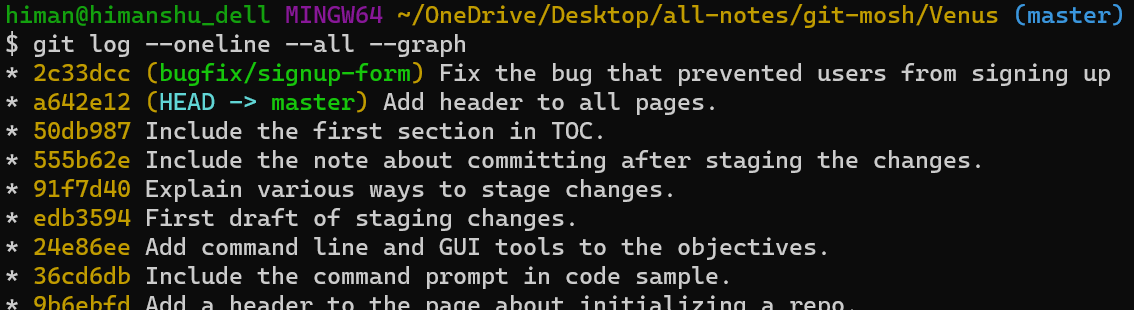
**Fast – forward Merges**:

Let’s see a real example of fast forward merge.

First let’s look at our log

git log --oneline --all --graph

Note: When we are using branches, it’s a good idea to always include --graph option. It gives a better representation of our branches and how they diverged.



As we can see our bugfix branch is one commit ahead of the master branch (*also we have a linear path here*), so if we start from bugfix we can get to master (*master has not diverged from bug fix*).

So to bring the changes from bugfix branch to master.

🡪 First we should be on the master and then,