**Introduction:**

The first thing we need to know to use GIT effectively is how to take snapshots of your project.

**Initializing a Repository**:

First thing we need to do is create a directory for our project.

*C:\Users\himanshu\Desktop\git-mosh*

mkdir moon

cd moon

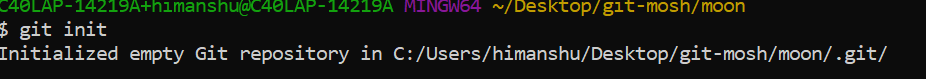
*C:\Users\himanshu\Desktop\git-mosh\moon*

I created a project directory called *moon* inside *git-mosh* folder, where we can have 10s or 100s of project files.

*The first time we want to add these files to a GIT repository, we have to initialize a new empty repository*.

$ git init

We get a message on our terminal with full path

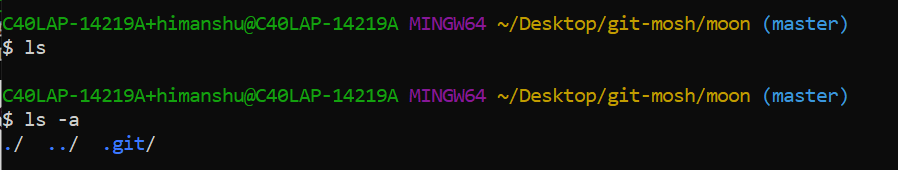


So .git is the subdirectory inside our project directory.

Note: By default this .git directory is hidden, because we are not supposed to touch it.

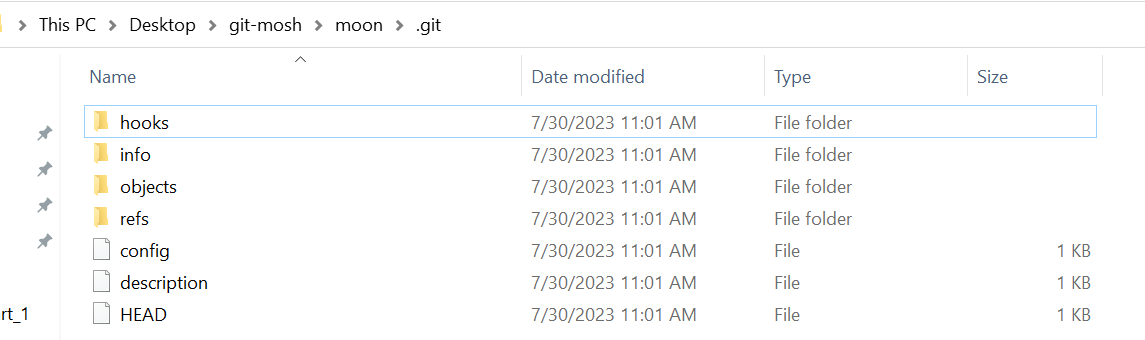
To view it,

ls -a



To open this folder in file explorer from terminal,

$ start .git



Here is our GIT directory or repository, where git stores information about our project history. It is not our business, this is just how git stores information (*that is why it is hidden*).

*If we corrupt or remove this directory we will lose our project history*.

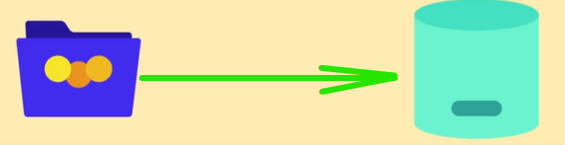
Note: If you want to have a colorful terminal in windows download posh-git.

**Git Workflow**:

Now we have a git repository, let us talk about basic GIT workflow, meaning what we do on daily basis when using git.



1. Here we have a *Project directory* and a *GIT repository* (*which is actually a subdirectory inside our project directory*).
2. Now everyday as part of working on various tasks we might modify one or more files and when our project reaches a state we want to record, we commit those changes in to our repository.



“*Creating a commit is like taking a snapshot of our project*”.

1. Now in GIT we have a special area or a special intermediate step that does not exist in most version control systems. It is called the ***staging*** area or the *index*.

*It is simply what we are proposing for the next commit or the next snapshot*.

So when we are done making changes, we *add the modified files to the staging area or the index, review our changes and if everything is good make a commit*.



Proposed snapshot will get permanently stored in our repository. So the staging area allows us to review our work before recording a snapshot.

1. If some of the changes should not be recorded as part of the next snapshot, we can *unstage* them and commit them as party of another snapshot.

That’s the basic workflow. Now let us walk through a *real example*.

1. Our Project directory is currently empty, so we add a couple of files here. Now if we are ready to record this state we use ***add*** command *to add these files to the staging area*.



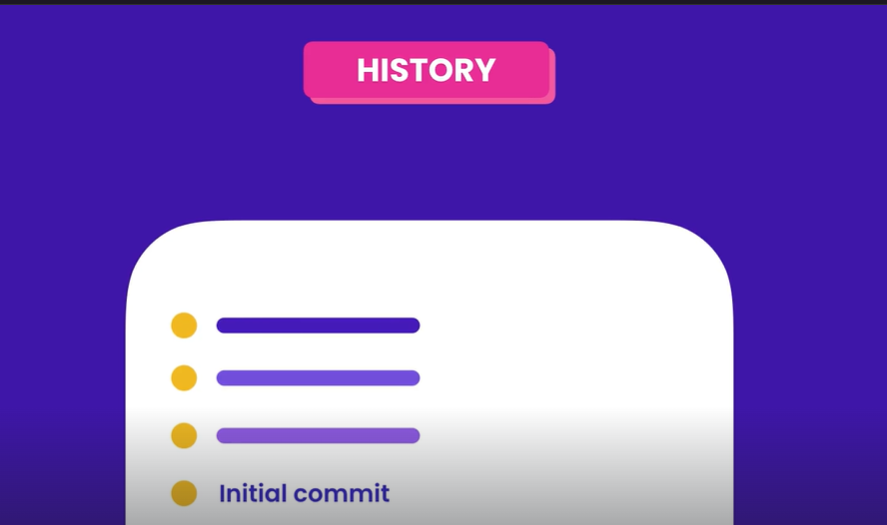
Now these files are in the staging area, which is the state we are proposing for the next commit.

1. Now we review these files and everything seems good. So we use ***commit*** command to permanently store this snapshot in the repository.



As part of this we supply a meaningful message (*like “initial commit”*) to indicate what this snapshot represents. This is essential for having a useful history.

1. As we fix bugs, implement new features or refactor our code, we make commit and *each commit clearly explains the state of the project at that point in time*.



Note: A common misconception about GIT is that *once we commit the changes, the staging area becomes empty which* ***is not correct***.

So after “initial” commit what we currently have in staging area is the same snapshot that we stored in repository. So this staging area is very similar to staging environment we use when releasing software to production.

*It is either a reflection of what we currently have in production or the next version that is going to go in production*.

Moving on…

1. Let us say we fixed some bug in *file1*, in our project directory.

Note that in staging area we have old version of *file1* because we have not staged the changes yet. So once again we use the ***add*** command to stage the changes.



Now what we have in staging area is the same content we have in our working directory.

1. So we make a commit to record this stage, making total two commits in our repository.



Also look at the commit message which is describing the bug that we just fixed.

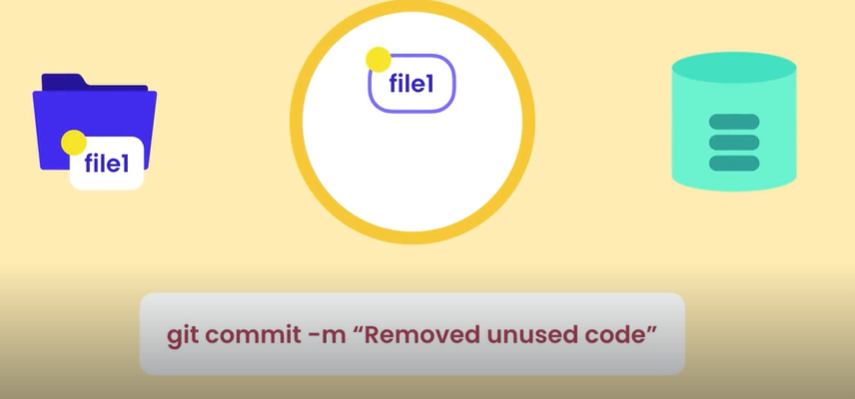
1. Let us say we no longer need file2 as it contains useless code. So we remove this file from our working directory.



But this file is still in our staging area. So once again we should use ***add*** command to stage this change. In this case *deletion*.

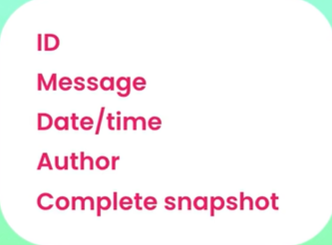
 🡨 Even though we are saying *add file2*, GIT knows that file2 is actually deleted. So it will delete this file from staging area as well.

1. Again we make a commit to permanently record this state.



Now we have 3 commits in our repository.

Note: Each commit contains a unique identifier that gets generated by GIT. *It also has information about what was changed, by who, when and complete snapshot of our project at the time it was created*.



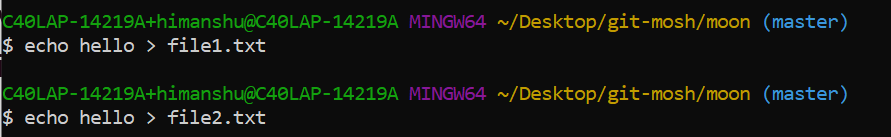
It does not store the deltas or what was changed, it stores the full content (*helps to quickly restore project to earlier state*).

Note: Some people might think that storing the full content in every snapshot wastes a lot of space, NO IT DOES NOT.

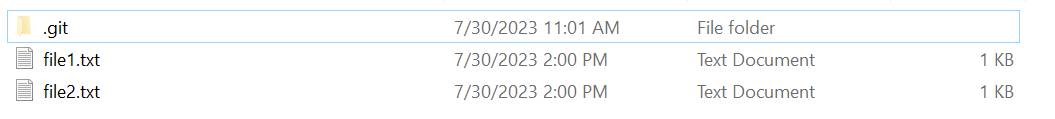
Because GIT is very efficient in data storage. *It compresses file content and does not store duplicate content*.

**Staging Files**:  
Let us add couple of files into our project.

We will use ***echo*** (*Linux command*) command to write content into a file.

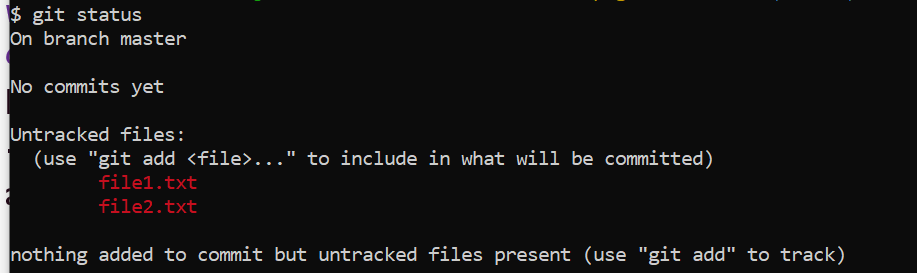


Two files are created inside our working directory.



These new files are not yet tracked by GIT, because *first time we initialize a git repository in a directory git is not going to automatically track your files*. So if you have 1000 files in your project, you have to instruct git to track them.

🡪 Use git status to *see status of working directory in staging area*.



Our files are indicated by *red* cause they are not in staging area yet. To add these files to staging area we use the git add command.

Here we can add a single file,

$ git add file1.txt

Or add multiple files separated by space.

$ git add file1.txt file2.txt

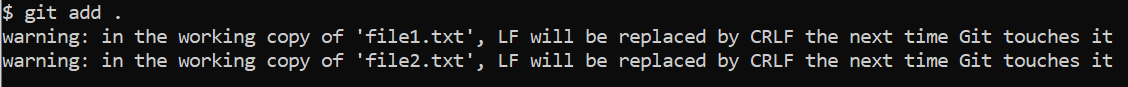
Or use a pattern using **\***, like add all files with .txt extension.

$ git add \*.txt

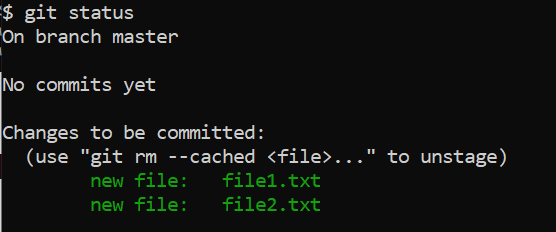
Or add the entire directory recursively using**.** (*Period*)

$ git add .

Note: Be careful with this because sometimes there are files that you do not want to add to your repository (*like large binary files or log files*).



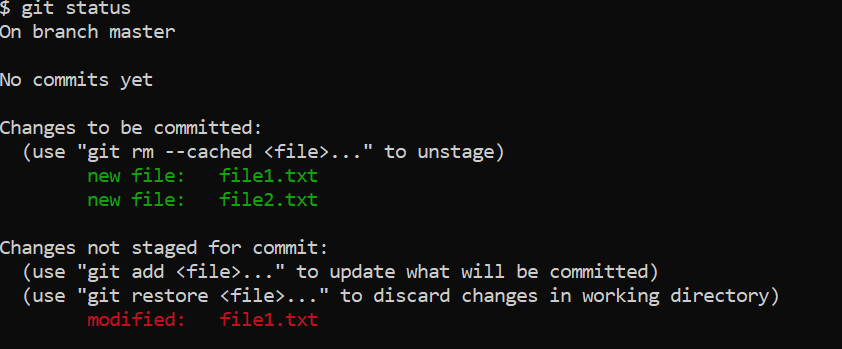
Now run git status again,

🡨Green means in staging area.

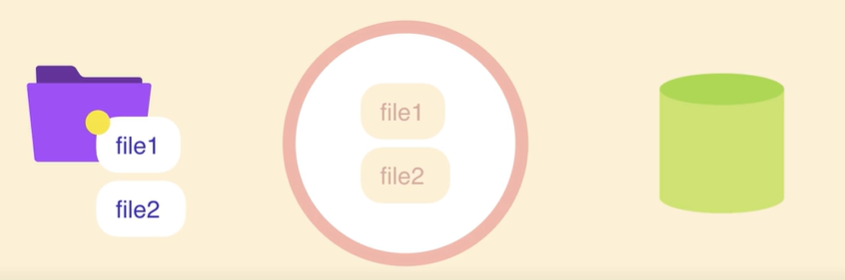
🡪 I will modify file1.txt by appending some text (*using >>*).

$ echo world >> file1.txt

And run git status again.



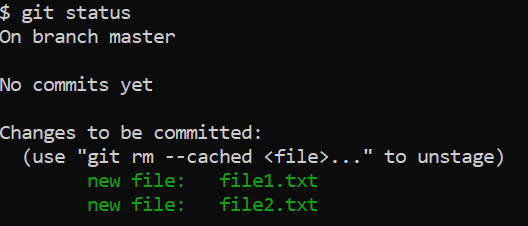
We already had two files in staging area as they are *indicated by green*. But now we also have one modified file in our working directory *indicated by red*.



In the staging area, we have the first version of file1. We changed this file after we added it to staging area.

So what we have in our working directory is the second version of this file (*it has changes which are not staged yet*).

🡪 If we git add file1.txt one more time,



We do not see any *unstaged* changes now.

**Committing Changes**:

Now, we have snapshot in the staging area, ready to be permanently stored in our repository.

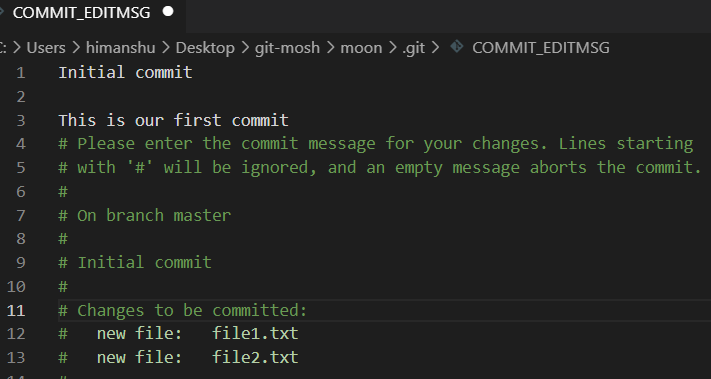
$ git commit –m “Initial commit”

Here *m* stands for message which is short description of what this snapshot represents.

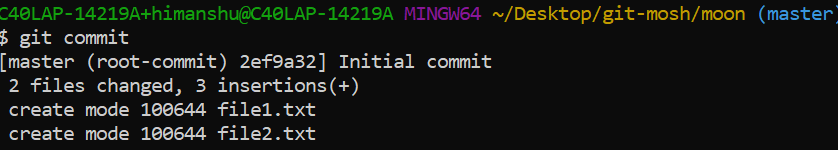
Sometimes short message is not enough and we might need to give more details about specific changes done in the file.

So just type $ git commit.

It will open our default editor with COMMIT\_EDITMSG file, which is stored in our git sub directory.



Save the changes and close the file.

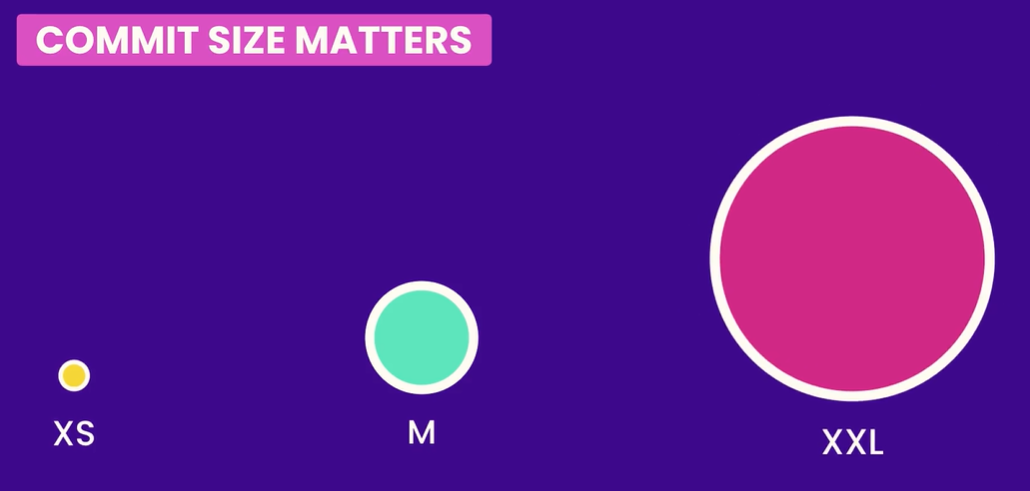


3 insertions, (*2 in file1 and 1 in file2*).

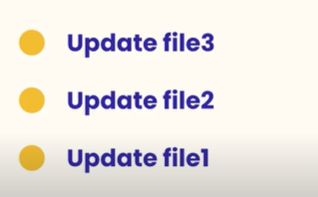
**Committing Best practices**:

Best practices of committing code,

1. Commits should not be too big or too small.



1. Do not make commits every time we update a file, it is just useless. We will end up with commits like these…



1. Commits should not be too big. We do not want to wait and implement a feature end to end before committing it. *Like waiting for 3 days to make a commit*.

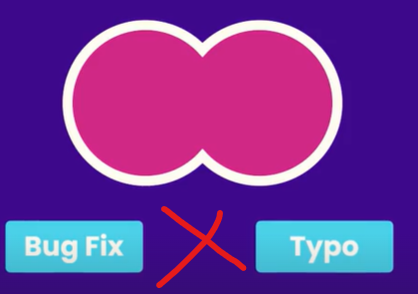
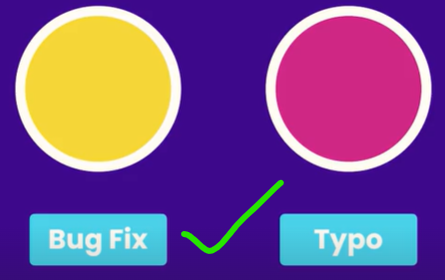
*The whole point of committing is making checkpoints as we go*.

1. *Commit often*. Maybe 5-10 times a day depending on type of work we do.

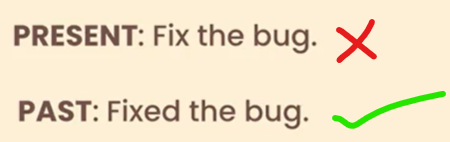


1. Each *commit should represent a logically separate chain set*. So do not mix things up.

For example, you are fixing a bug and then you accidently find a typo in your app. Do not commit both changes in one go. Have two separate commits

1. Follow the habit of creating *meaningful commit messages*. It is because all these messages will come up in history and if they are cryptic they are not going to be helpful to us or our team members.
2. Some developers use the present tense in their commit messages *use past tense* instead. So instead of fix the bug say fixed the bug.



**Skipping the Staging Area**:

A common question a lot of beginners have “*Do we always have to stage the changes before committing them?*”

Answer is NO, we can skip the staging.

Note: Do this only if you are 100% sure that your code does not need to be reviewed.

Let us change something in file1.

$ echo test >> file1.txt

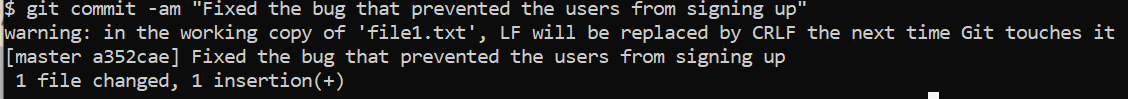
Now instead of git add, we will run

git commit –a –m “message”

Where *–a* stands of all modified files.

We can also write it like this,

$ git commit -am "Fixed the bug that prevented the users from signing up"



This is how we can skip the staging area.

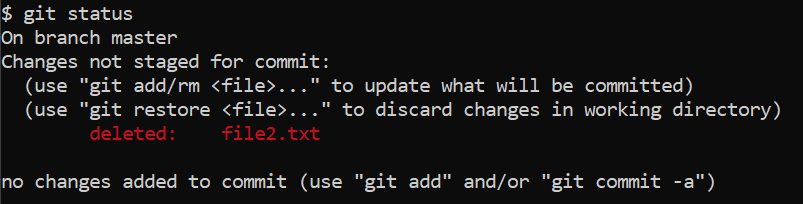
**Removing Files**:

Imagine we do not need *file2* in our project because it contains unused code.

*To remove file*,

$ rm file2.txt

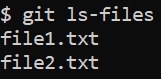
We see git status



We have one change which is not staged for commit. Because file2 is removed from working directory but it is still in staging area.

*To view files in staging area*,

$ git ls-files

🡨File2 is still in staging area.

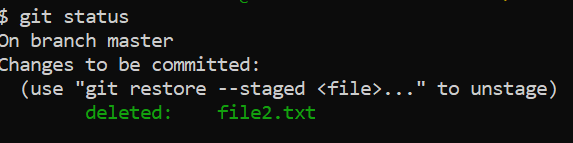
We know that every time we make changes, we have to stage those changes using ***add*** command, to stage this change (*or deletion more accurately*)

$ git add file2.txt

View files in GIT repo using ls-files again,

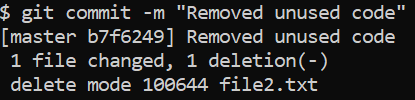
🡨 File2 is no longer in staging area.

Now, we see GIT status again with one change ready to be committed as indicated by green.



Finally we commit this change,

$ git commit -m "Removed unused code"



To remove a file we need to remove it from both our working directory as well as the staging area.

Note: Since this is a very common operation, GIT gives us a command that performs both of these steps in one go.

$ git rm file2.txt file1.txt

Here instead of using standard UNIX command rm to we use ***git rm*** and give different file names that we want to remove (*can also use pattern like \*.txt as argument*).

*This will remove file from both working directory as well as staging area*.

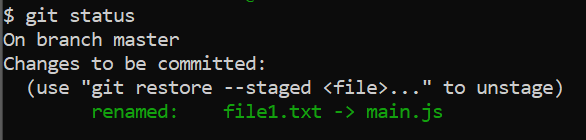
**Renaming or moving files**:

This is also a standard operation, so GIT gives us a ***git mv*** command.

Let us rename our file1.txt to main.js.

$ git mv file1.txt main.js

And check the git status,



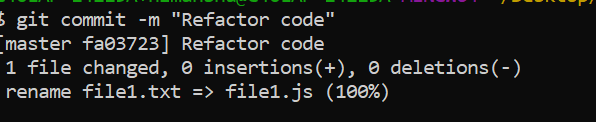
So when we use *git mv* command, changes are applied in both working directory as well as staging area.

Let us run *git mv* one more time,

$ git mv main.js file1.js

And do git commit,

$ git commit -m "Refactor code"



Note that we see 0 insertions because we have not added any new lines.

**Ignoring Files**:

In almost every project, we should tell GIT to ignore certain files and directories (*for example log files*).

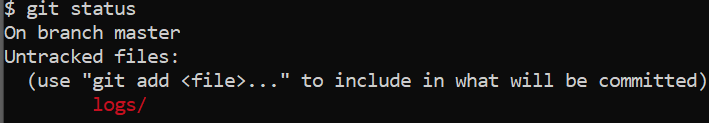
So in our working directory, let us add a new directory called *logs*. using *mkdir* command.

$ mkdir logs

Use echo to create a file and write some content into it.

$ echo hello > logs/dev.log

Check git status.



It is saying that we have an untracked directory called *logs*. But we do not want to add it in staging area to avoid being tracked by git.

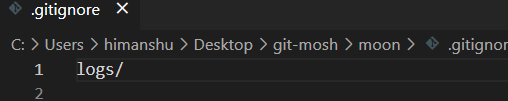
To prevent this, we create a special file called .***gitignore***. This file has no name just an extension and *it should be in the root of your project*.

Let us add our logs directory into *gitignore*,

$ echo logs/ > .gitignore

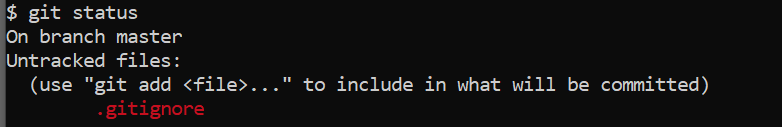
We can open this file in VS code,

$ code .gitignore



In this file we have a single entry *logs/* (*/ indicates a directory*). We can mention files here as well as a pattern.

If we run git status again, git no longer notice *logs* directory, instead it has a new untracked file called *.gitignore*.



Now add and commit this,

$ git add .gitignore

$ git commit -m "Add gitignore"

*Which files we cannot ignore*?

If we accidently add a file in our repository and then later add to gitignore, GIT is not going to ignore that.

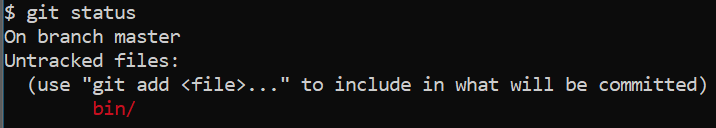
Let us make a new directory called *bin* (*imagine it contains our compiled source code*),

$ mkdir bin

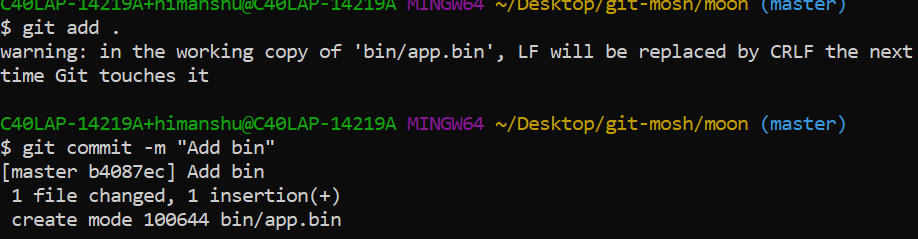
Create a new file into this directory and called *app.bin*.

$ echo hello > bin/app.bin

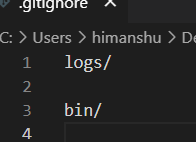
Git status:



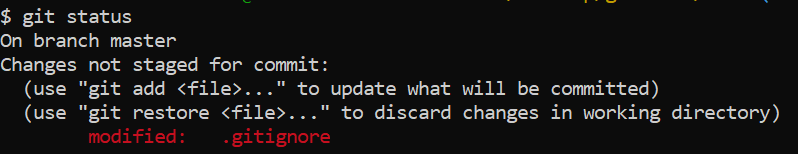
Now, we need to accidently commit to our repo,



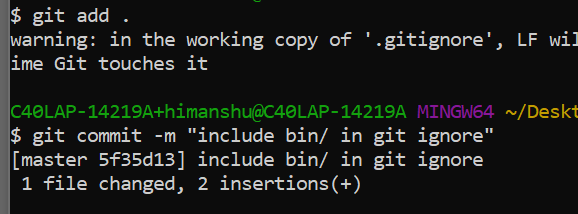
Now add bin directory in our gitignore file as well,



Git status shows we have modified gitignore

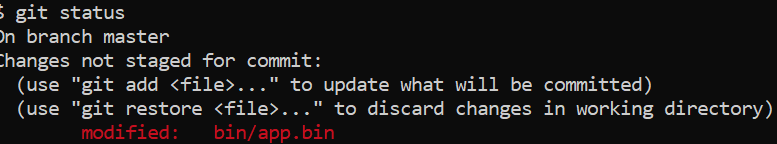


Add and commit,



Moment of truth, *what will happen if change something in bin/ directory*?

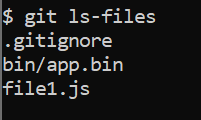
$ echo helloworld > bin/app.bin



Git says this file is modified, which is not what we want. To solve this problem we need to remove this directory from staging area.

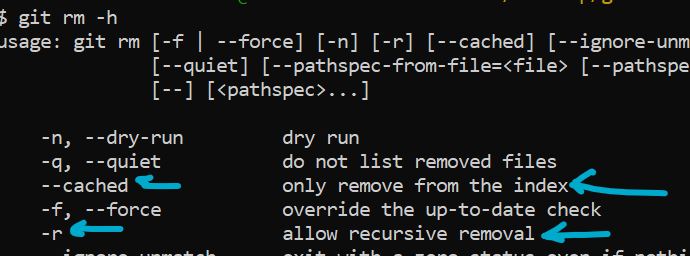
To view the files in staging area,

$ git ls-files



We can remove files from both working directory and staging area using *git rm* command. But in this case *we only want to remove this file from staging area and keep it in working directory*,

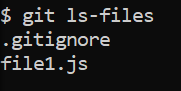
Note: Use git rm **–h** for help on git remove command.



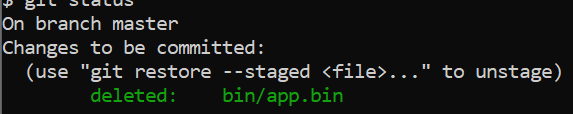
There is --cached command with which we can remove files only from index (*index is legacy name for staging area*). Other option is –r which is for recursive removal (remove entire directory from staging area).

$ git rm --cached -r bin/

Now directory is removed from staging area,

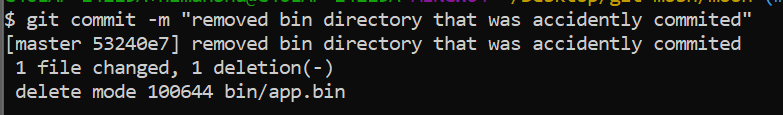


Check the status,



We have this change ready to be committed.

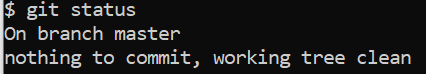
$ git commit -m "removed bin directory that was accidently commited"



From this point forward, git is no longer going to track changes in this directory, so if we change something now in *app.bin*,

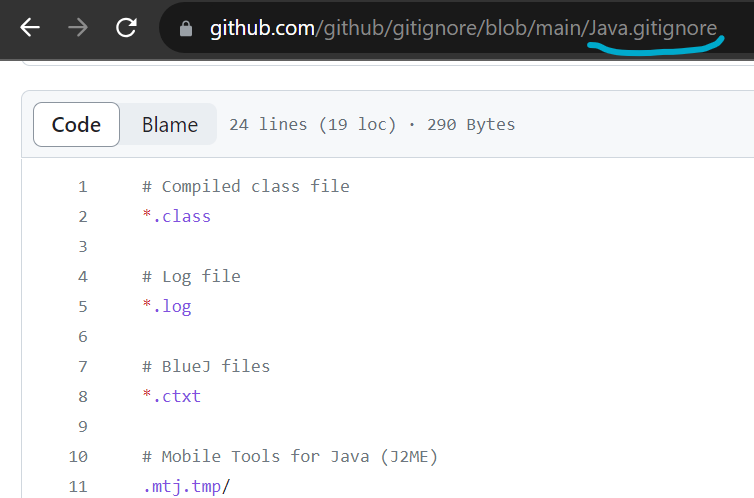
$ echo test > bin/app.bin

And check the status,



This is how we ignore files and directories in GIT.

Note: If you head over to <https://github.com/github/gitignore>, you will see various gitignore templates for different programming languages.

🡨 Java gitignore

**Short status**:

So we have learned how to get the status of the working directory and staging area using *git status* command.

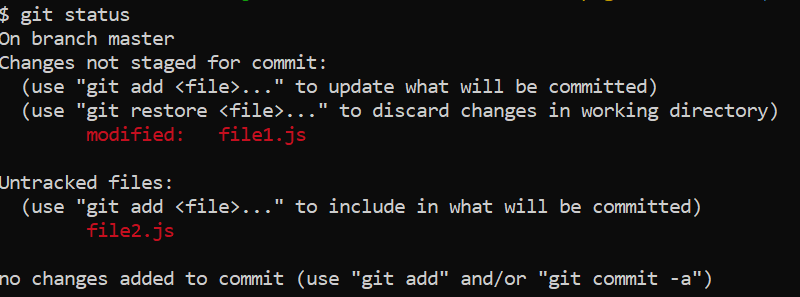
The output of this command is very comprehensive and wordy. So as an additional argument we can provide –s (*for short*) status.

For this demo *modify an existing file and add a new one*,

$ echo sky >> file1.js **M**odify file

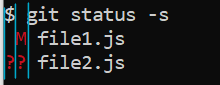
$ echo sky > file2.js **A**dd a new file

If we do git status now, we see a very comprehensive but wordy output on terminal,



Now let us run,

git status –s

🡨 Here we have two columns.

*Left column* 🡪 Staging area

*Right Colum* 🡪 Working directory

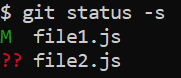
We have modified file1.js that is why we have M in right column which is the working directory. Changes are not committed therefore we have nothing in staging area.

File2 is a new file (*untracked*) therefore (??) in left and right.

🡪 Stage the modified file,

$ git add file1.js

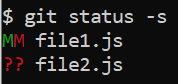
And check short status,

🡨 For file1 we have M on left (*staging area*) now.

If we change file1 again,

$ echo ocean >> file1.js

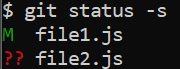
And check status,



It means we have some changes in staging area (*left*) but also some additional changes in our working directory (*right*) that are not added to staging area.

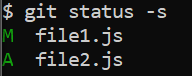
If you run *add* command one more time,

$ git add file1.js

 🡨 Now all the changes we had in working directory are now in staging area.

Let us add file2 to staging area as well,

$ git add file2.js

🡨 File1 is modified (M) while file2 is newly added (A).

**Viewing Staged and Unstaged Changes**:

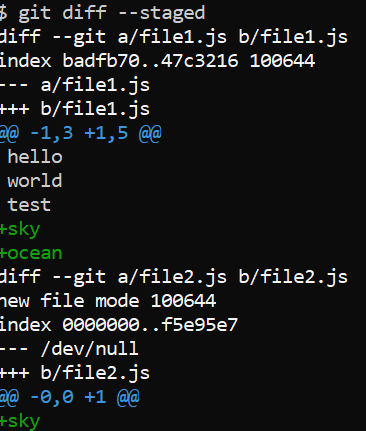
So we have staged a couple of changes, now before committing what we have in the stage. We need to review our code because we do not want to commit broken code to our repository.

*As a best practice always review what we have in staging area before making a commit*.

*How to see exact lines of code that have been* ***staged***?

Use *git* ***diff*** command.

🡪 To see what we have in staging area that is going in next commit add -- staged in argument.

****

Let us understand its output.

1. *diff* utility was called with arguments

diff --git a/file1.js b/file1.js

Here we are comparing two copies of the same file

a/file1.js is the old copy which we had in last commit

b/file1.js is the new copy which we have in staging area.

1. Below we have,

index badfb70..47c3216 100644

It is some metadata.

1. Then we have,

--- a/file1.js

+++ b/file1.js

Changes in old copy are indicated by ---

And changes in new copy indicated by +++

1. Next we have a header,

@@ -1,3 +1,5 @@

Which has information about what parts of our file are changed. Git diff does not show entire file but rather divides all the parts that we have changed into chunks and all these chunks have a header.

@@ -1,3 +1,5 @@

hello

world

test

+sky

+ocean

-1, 3 🡪 in the old copy starting from line 1, 3 lines have been extracted and shown here (*hello, world, and test*).

+1, 5 🡪 in the new copy starting from first line, 5 lines have been extracted and shown here. Lines prefixed with + sign are added the new copy (*sky, ocean*).

1. After that we have another call to diff utility.

diff --git a/file2.js b/file2.js

This time we are comparing two copies of file2.

1. --- /dev/null shows we have no old copy because it was an entirely new file.
2. That is why we have @@ -0,0 +1 @@ , starting from 0 index 0 lines have been extracted.

*How to view changes in our working directory that have* ***not been staged*** *yet*?

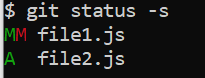
Run *git diff* without any arguments.

This compares what we have in working directory with what we have in staging area.

Note: If you do not see any output of this command, it means you have staged all the changes in working directory.

For this demo change the content of *file1.js as Hello to Hello world*.

Now if we run short status,



We see we have changes in working directory now.

Run *git diff*,



a/file1 is in staging area while b is in working directory.

**Visual Diff Tools**:

There are multiple diff tools available in the market like,

🡪 KDiff3

🡪P4Merge

🡪WinMerge (windows only)

🡪 VSCode (*will use for this demo*)

1. First we have to tell git that we will use vscode as our default diff tool, so we have two set two configuration settings.

$ git config --global diff.tool vscode

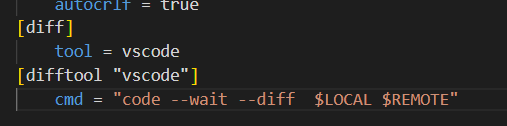
1. Next we tell git how to launch VS code,

$ git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"

LOCAL and REMOTE are placeholders for the old and new copies of the file.

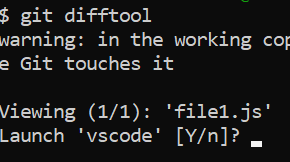
1. To verify if our settings are updated as required, open config file

$ git config --global -e

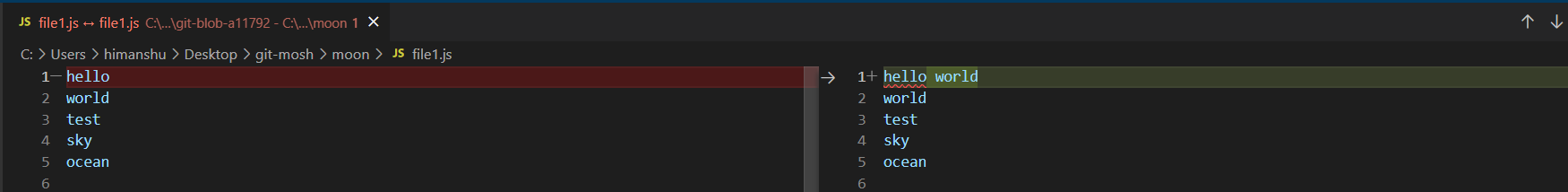


1. Now to run our diff tool, run this command,

$ git difftool (*to view unstaged changes*)



We have modified only a single file, so showing 1/1.

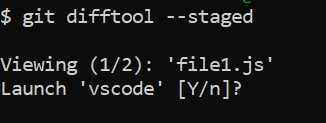


On left we have old copy in staging area and on right we have new copy in our working directory.

It is much easier to see the changes

1. Now to view staged changes

$ git difftool --staged



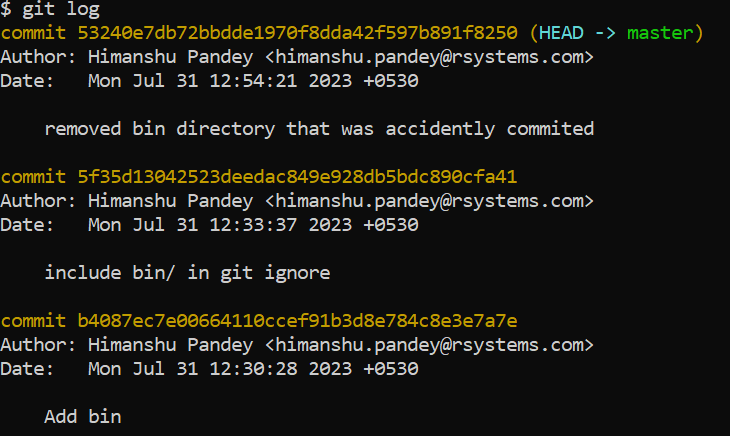
Viewing 1//2 because two files have been affected in staging area.

**Viewing the History**:

We have made a few commits so far, but *where are these commits*?

We use the ***git log*** command to view our history.

$ git log



Here are all the commits we have created sorted from latest to the earliest.

Each commit has a unique identifier (*a 40 character hexadecimal string*) that automatically generates for us.

(HEAD -> master) Here master is the main branch or the main line of working in GIT (*in some other version control systems it is called trunk*).

In GIT we can have multiple branches, so we can work on multiple features or multiple bug fixes in parallel and then combine our code.

HEAD is the reference to the current branch. So this is how GIT knows on which branch we are currently working on.

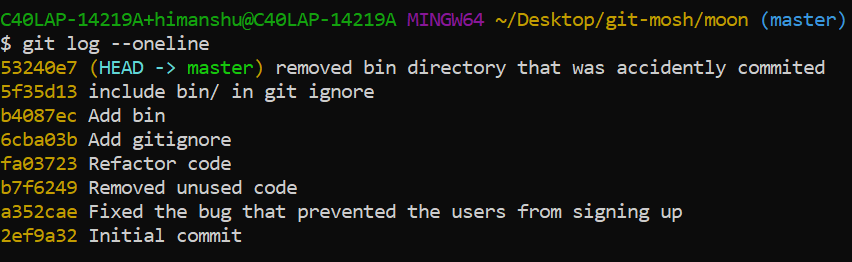
For each commit we can see author (*with name and email*) and date – time when this commit was created as well as one line description.

Note: Use ***space*** to go to next page while browsing through log history and ***Q*** to quit.

Git log has few interesting arguments like,

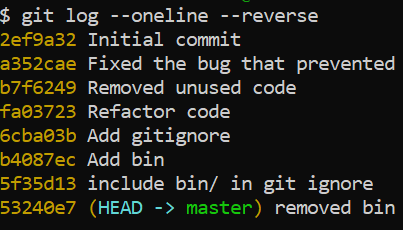
🡪 To see the *short summary of commits*,

$ git log --oneline



🡪 To *reverse the order of commits*,

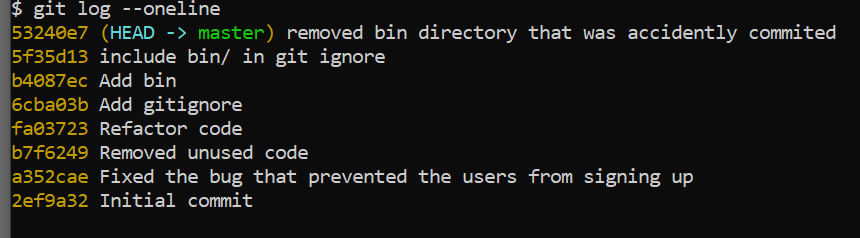
$ git log --oneline --reverse



**Viewing a Commit**:

So viewing the list of commits is great, but what *if we want to see what exactly we have changed in a given commit*.

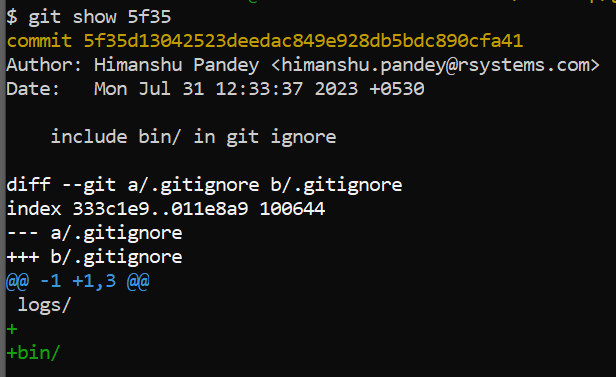
We use the ***git show*** command.



We specify the commit we want to inspect. There are two ways to reference a commit.

🡪 We can reference it using its unique identifier.

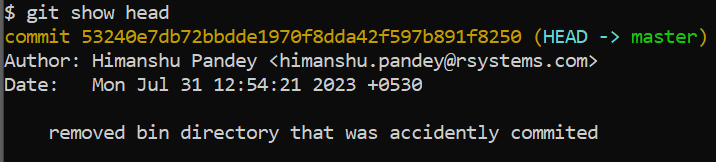
$ git show 5f35d13 or $ git show 5f35 (*we can also use fewer characters*, *if not repeated elsewhere*)

🡨We see the *differences in commit*.

🡪 Another way is to use the ***head*** pointer.

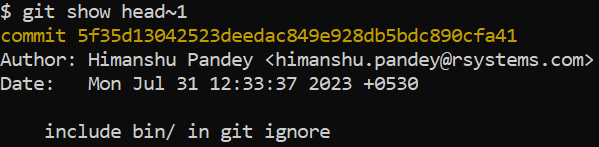
If you want to see last commit type,

$ git show head



If need to see first previous from last use ***tilt*** *(****~****)* sign,

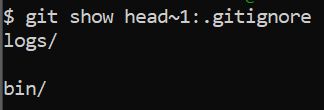
$ git show head~1 (*similary second previous will be $ git show head~2*)



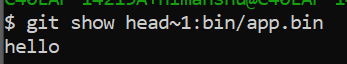
What if *we do not want to see differences but the final or exact version that is stored in this commit*?

🡪 In the same previous command git show head ~1, supply one more argument (*the exact file path of file*) after *colon***:** …

$ git show head~1:.gitignore



$ git show head~1:bin/app.bin (*path in case of subdirectory*)



These are the exact versions stored in this commit. Now we know that each commit contains a complete snapshot of our working directory, not just changes.

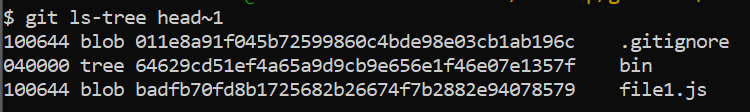
But when we run the show command, we only see the differences.

What *if we want to see all the files and directories in a commit*?

For that we use a different git command, it is called ***ls-tree***.

*Tree is a data structure that represents hierarchical information. So in a tree we can have nodes and these nodes can have children. These children can be files in other sub directories*.

$ git ls-tree head~1 (*all files, directories in this commit*)

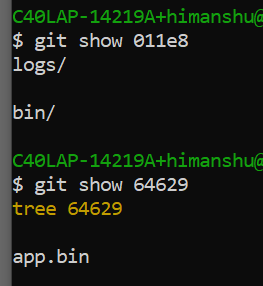


Note: We have unique identifiers generated based on the content of this file. Here *files are represented as blob* and *directories are represented as tree*.

All these objects are stored in git database, using show command we can easily view that object.

$ git show 011e8 (*blob*)

$ git show 64629 (*tree*)

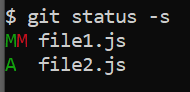


These GIT objects we can view using ***show*** command,



**Unstaging files**:

We know that we should always review the stuff we have in staging area before making a commit.



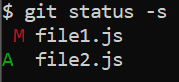
Suppose that changes that we made in file1 should not be committed (*in left staging area*) as they are logically part of a different task.

So in this case we will *undo the git add* operation by using ***git restore*** command. With this command *we can easily restore files in different environments* (*working directory or staging area*).

$ git restore --staged file1.js

We have restored *file1* from staging area, hence the --staged argument and lastly the name of the file (*we can use \* for patterns or****.*** *If need to restore entire staging area*).

Now if we check git status again,



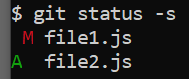
*File1* is removed from the staging area and now only in working directory.

*How restore command work behind the scenes*?

The *restore command essentially takes the copy from the next environment*. So in case of staging environment the next environment is last commit (*what we have in repository*).

So when we restored file1 in the staging area, GIT took the last copy of this file from the last snapshot and put it in the staging area.

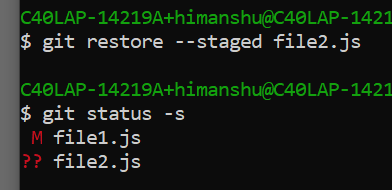
Below file2 is the new file that we have added in the staging area but it does not exist in the last commit.



What will happen *if we restore this file which do not have a copy in the repository*?

GIT is going to remove this file in staging area and take it back to its previous state (*which is a new untracked file*).

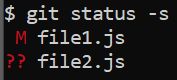
$ git restore --staged file2.js



**Discarding local changes**:

There are times when we have some code in our working directory that we want to throw away.

We can *discard the local changes* using the *restore* command.

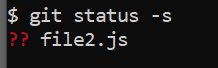


Here we have some local changes in file1. *To undo all local changes*

$ git restore .

When we run this command, GIT take the version of this file in the next environment which is our staging environment and copy into our working directory.

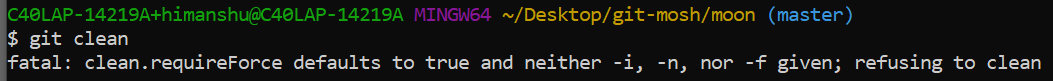
Check status,



*File1* is removed from working directory as well but *file2* is still here because this is a new untracked file. *Since GIT does not know where to get a previous version of this new untracked file it cannot restore or undo it*.

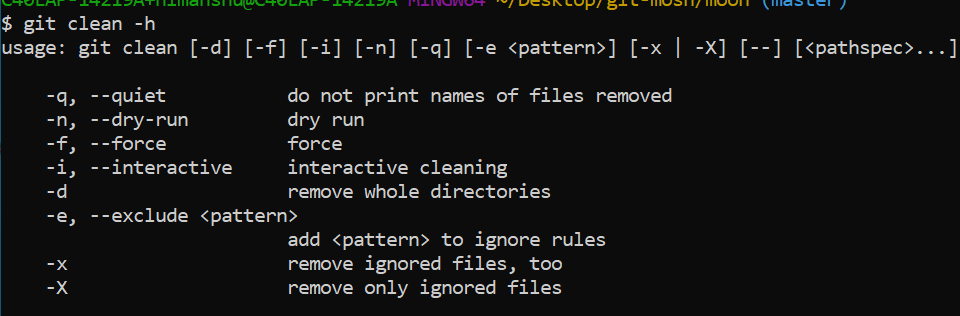
To remove all files even the untracked files use ***git clean***,

$ git clean



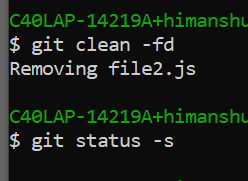
By default we get a fatal error refusing to clean, so look at the clean command help,

$ git clean -h



Quite often we use,

$ git clean -fd



*File2* is gone now.

**Restoring a file to an earlier version**:

Once git tracks a file, it stores every version of that file in its database. Which means if we screw things we can always restore a file or a directory to a previous version.

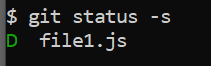
In this demo we will delete a file and then restore it.

$ git rm file1.js

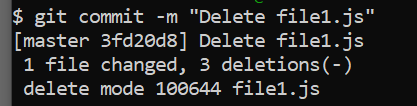
This command will remove file1 from both staging and working directory.



Short status

🡨 In staging area we have deleted file, so let us make a commit.

$ git commit -m "Delete file1.js"



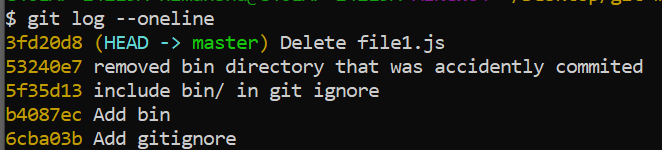
Now there are *2 ways to restore the deleted* file,

1. *Undo or reverse the last commit*:

Will discuss in next section

1. *Restore the file to previous version*.

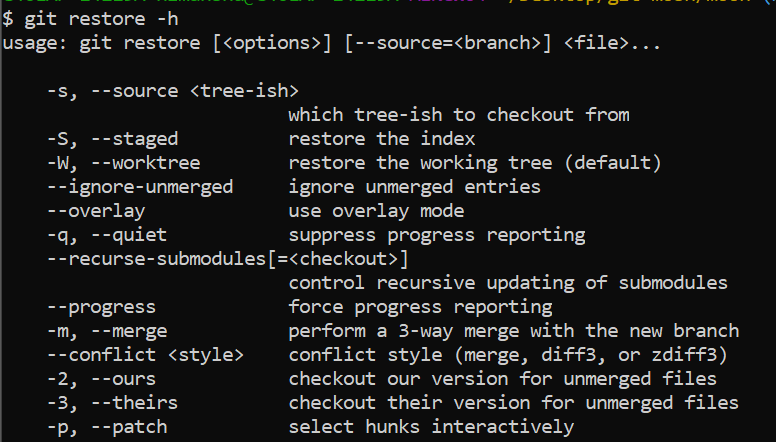
Look at the history, $ git log --oneline,



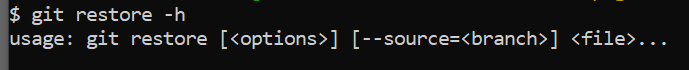
We want to restore *file1* to the commit before the last commit.

Let us look at this commands documentation

$ git restore –h



The ***restore*** command takes three types of arguments.



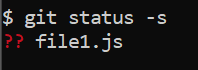
We can supply bunch of *options*, a *source* (*if not specified then git will restore the file from the next environment / area*)

*For example* if the file we want to restore is in working directory then git will restore it from the staging area; And if the file is in staging area git will restore it from the last commit (*so the next environment by default*).

In this case we will change the default behavior,

We will *restore file from the commit before the last one*.

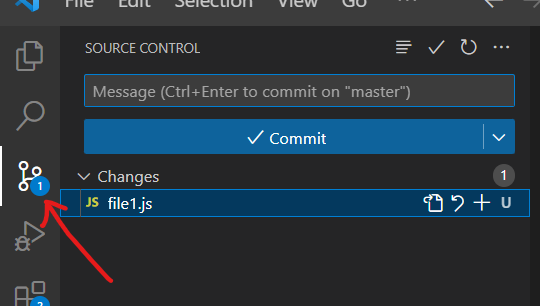
$ git restore --source=head~1 file1.js

🡨 Now we have a new untracked file.

This is how we can restore file to a previous version.

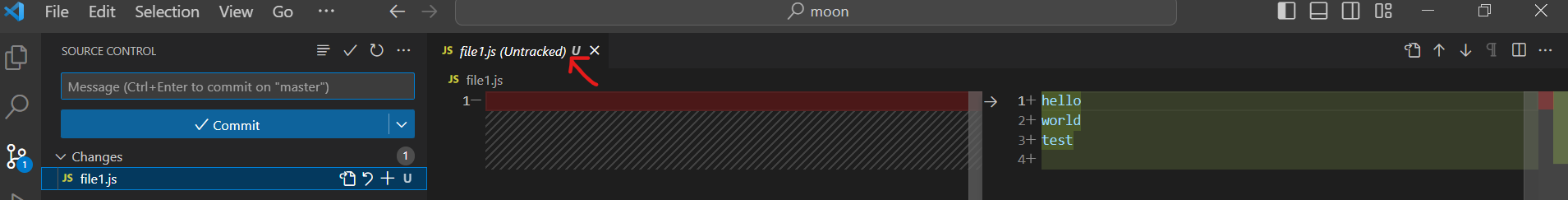
**Creating Snapshots with VS code**:

Now let us look at the GIT tools built in VS code.



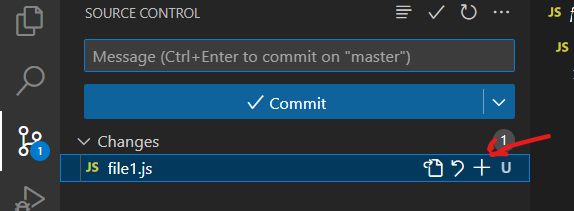
On left side we have *the source control panel*, where we can see all the changes in our project. It is very similar to output of status command (*we can see changed files and changes introduced in those files*).

When we click on the *file1* under changes, we get *diff* of that file.

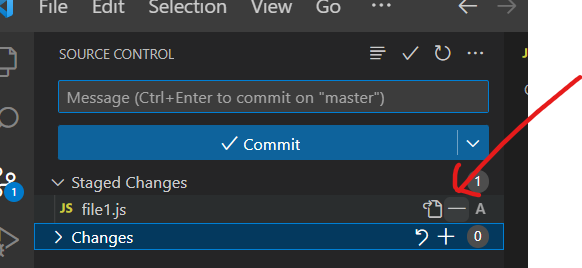


Here we are dealing with a new untracked file that is why we have no code in the left side.

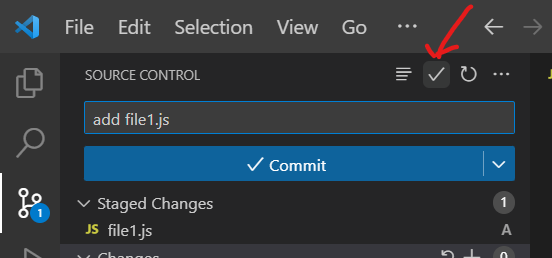
We can *stage this file by clicking* on **+** icon on file name,



If some mistake is made, we can unstage changes with **–** icon,

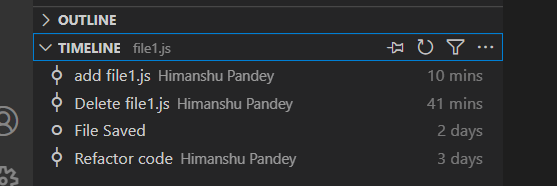


To commit the changes, write the message and commit

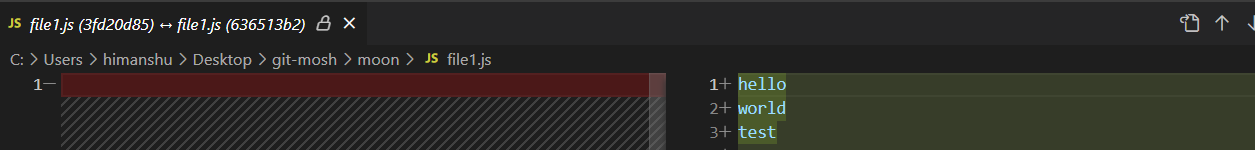


Note: To see all the commits applied to the file in VS code,

Click on file in file explorer and open *timeline* below,



If you click on any commit here, you will see the changes introduced in that commit,



Note: To look at our history in VS code we need another plugin