Git Basic

Hamid Semiyari

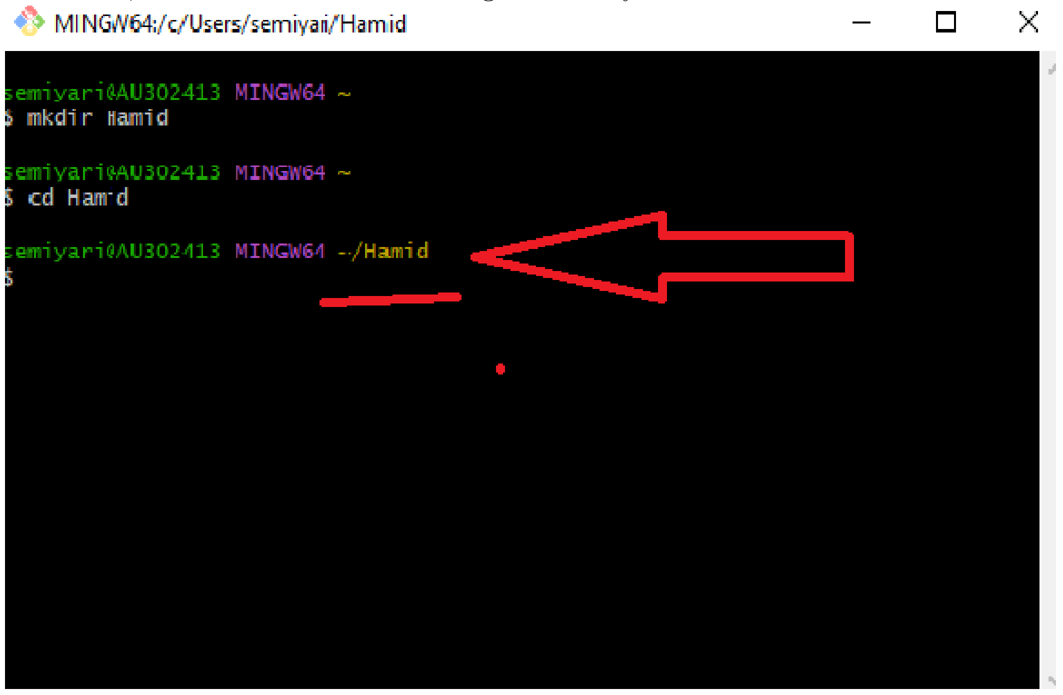
A Quick Introduction to Git and GitHub. Click here

GIT COMMANDS

- How do we take snapshots of our project?

1. Create a directory for our project. So Let us create a directory called `Hamid`
   You need to type

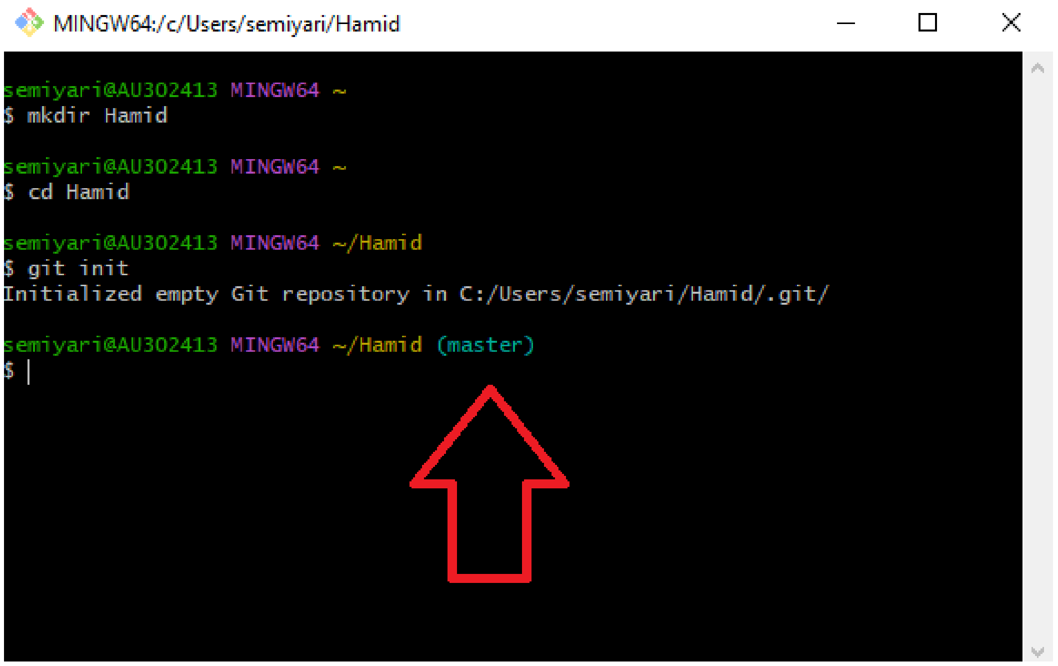
   - `mkdir Hamid` , where `mkdir` means "Make Directory" and go to this directory.
   - `cd Hamid`, where `cd` stands for "Change Directory". Now we can have our files in it.



```
MINGW64:/c/Users/semiyari/Hamid

semiyari@AU302413 MINGW64 ~
$ mkdir Hamid

semiyari@AU302413 MINGW64 ~
$ cd Hamid

semiyari@AU302413 MINGW64 ~/Hamid
$
```

2. We want to add files to a Git repository. We have to initialize a new empty repository.

   - So we type `git init`
   - The `git init` is a one-time command you use during the initial setup of a new repo. Executing this command will create a new `. git` subdirectory in your current working directory.
   - Running git init. in an existing repository is safe. It will not overwrite things that are already there. The primary reason for rerunning git init is to pick up newly added templates (or to move the repository to another place if –separate-git-dir is given).



```
MINGW64:/c/Users/semiyari/Hamid

semiyari@AU302413 MINGW64 ~
$ mkdir Hamid

semiyari@AU302413 MINGW64 ~
$ cd Hamid

semiyari@AU302413 MINGW64 ~/Hamid
$ git init
Initialized empty Git repository in C:/Users/semiyari/Hamid/.git/

semiyari@AU302413 MINGW64 ~/Hamid (master)
$ |
```

- You will see a pointer to the current branch (master or main)

Now we have initialized an empty Git repository in our directory.
Inside our directory, we have a sub-directory called `.git`. By default, this sub-directory is hidden because you are not supposed to touch it.
Type `ls` it will provide all files and subdirectories in your directory. You don't see anything. We have not added any files yet and the subdirectory `.git` is hidden.

If you type `ls -a` where "a" is short for "all", you will all files as well as the `.git` subdirectory.

- Windows: You can open this with Windows Explorer or file explorer.
  - Type `start .git`

- Mac: You can open this with "finder".
  - Type `open .git`
- Linux: Type `xdg-open .git`

This is where stored information about our project history. So, we have directories like hooks, info, objects, and references. This is how git stores information. That is why this directory is hidden. If you remove or crop this directory you are going to lose your project history. If you type `rm -rf .git` Now you do not have a git repository anymore. Since we have removed it we need to initialize it again. `git init`

- Note that deleting the . git folder does not delete the project files and folders. It just removes the functionality of Git from the project.
- **Question: What does "hook"mean?**
  - [Git hooks](https://www.atlassian.com/git/tutorials/git-hooks#:~:text=Git%20hooks%20are%20scripts%20that,in%20the%20development%20life%20cycle.) are scripts that run automatically every time a particular event occurs in a Git repository.
- `ls` it will provide a list of all visible files and sub-directories in your directory
- `ls -a` It will provide all files as well as hidden
- `ls -la` It will provide all files in the directory that start with dot
- We created our first git project and we initialized it and we saw that it created a `.git` directory inside of our folder (directory).
- **Dot files are invisible files.** How to look at the inside of `.git`? Type `ls -la .git` The result of these codes gives us all the files that Git uses to do all the tracking. We are not using any of these files except the `config` file. Type `cat .git/config`, where the `cat` is short for concatenation, and `.git/config` shows you the content of the file. Once again if you remove or delete `.git` Git will be removed from the project. It will no longer be tracking. There will be no tracking information left anywhere on your hard drive.
  Now we need to talk about basic Git WorkFlow

GIT WORKFLOW

THE THREE TREES

Lots of other version controls are Two Trees architecture. “Working Directory” and “Repository”. Git has Three Trees. “Working Directory”, “Staging Area, or Index” and “Repository.”

- Imagine we have a file “A” in the “working directory” (v_1). We use the `git add` command to move it to the “staging directory”. Now we use `git commit` to pushing it to the “repository”.
  Now repository has the same file and it is the same version as it is in our staging directory and working directory.
- Now we have made some changes to “A”. Let us call it “B”., now in the working directory we have “B” (v_2). Let us add it to the staging index and push it to the repository.

- Now the repository has two sets of changes “A” and “B”.

- The same process, Let us do some changes in the working directory and have file “C” (v_3). These are typical workflows, we are going to be using to make `commits`. You may use the `git log` to view those commands. To see what were the changes between each one.
- Git does not store duplicate content and also it compresses the content. So, it does not take as much as store you may think.

HASH VALUES

Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

- We have shown each change with “A” “B” and “C”.

- Git generates a “Checksum” for each change set.

- A checksum is a number that is generated by taking data and feeding it into a mathematical algorithm. So, the checksum converts data into a simple number. **The same data is always equal to the same checksum**.

- The checksum is used to guarantee the data integrity.

- The Data Integrity is fundamentally built into git.

- So, each "HASH value" is unique and directly tied to the contents that are inside of it.

- The algorithm that git is used is "SHA-1" (HASH algorithm).

- Git also does something else with data integrity. In addition use the code in each of snapshots (A, B, and C). It also uses "Metadata". That means you CAN NOT change the commit message or commit author or parent of the commit without changing its SHA value.
  - **Question: Are the checksum and hash values the same in git?**
    - In Git, checksum and has values are both used to verify the integrity of data, but they differ in their purpose.
      A checksum is used to detect accidental changes, while a hash value is calculated from a block of data using a *cryptographic hash function.* It is used to create a unique digital fingerprint of data, which can be used to verify the authenticity and integrity of data. Hash values are secure against malicious changes, as it is difficult to create a file with a specific crystallographic hash. So, hash values detect both accidental and malicious changes.

EXAMPLE

Let us add a couple of files to our project. We use a standard unix command called `echo`. We use it to write content to a file. Let us write "hello".

- Type `echo hello > file1.txt`. where 'echo' means to write "hello" and `>` means "to" the file "file1.txt".
- Type `echo hello > file2.txt`.

  - Please note that git does not automatically track your files so we have to instruct git to track them.

  - Working Directory is the only Tree in git architecture that does not require a git command to move file changes to that tree.

- Type `git status` to get the status of the **Working directory**. "Working Tree is clean" means the working tree is matches with what in the repository.
  - files are in **on branch master**
  - **No Commit** yet
  - We see that both of these files are in red font. which means they are not tracked yet.
  - They are not in the **staging area**. So, we need to type
    - `git add file1.txt` one by one or we can have multiple files separated by space

    - `git add file1.txt file2.txt`.

    - We may use `git add .` this stage **all** files (that are not in the `.gitignore`) in the entire repository. We have to be careful with that because there might be files that you do not want in your repository.

    - We also can use patterns such as `git add *.txt`. It will stage all files that have extension txt.

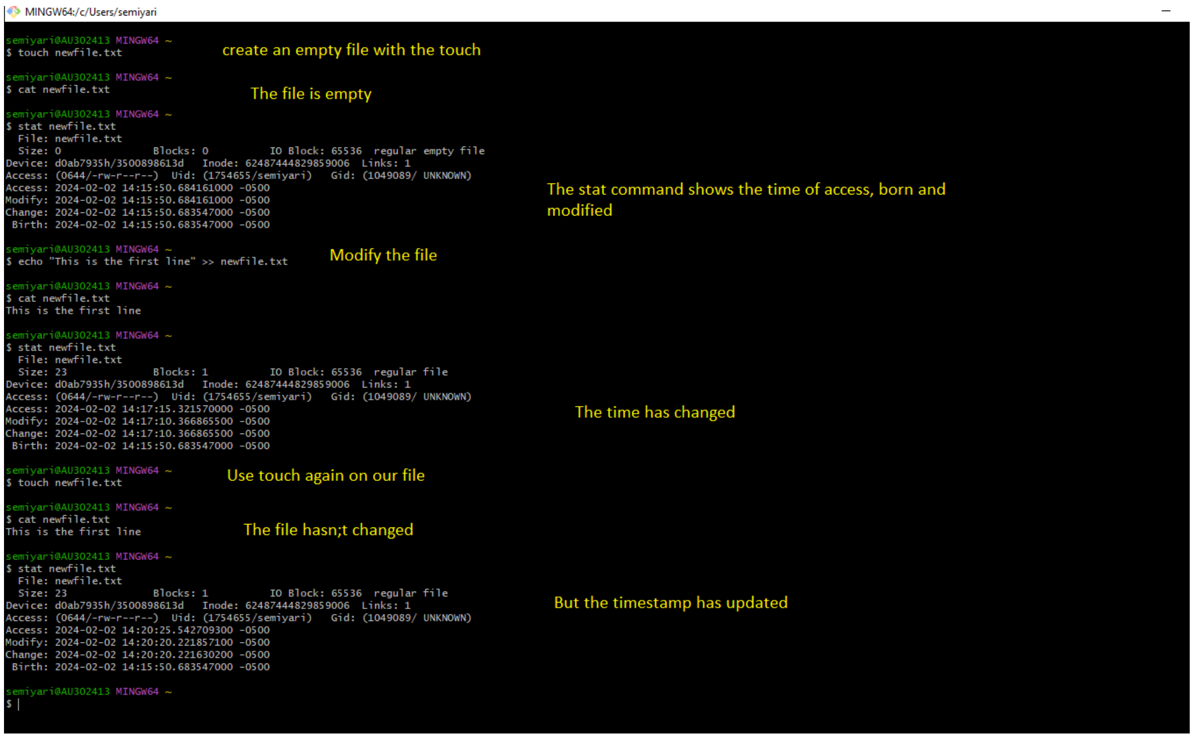
  If you type `git status` you will see those two files now in green. Which it means they are in the staging index.

`touch` and `stat` command

The Linux file system stores a wealth of metadata about every file and directory. This metadata includes data such as when it was last accessed or modified. - 1. `touch` command + This command can create empty files. More importantly, it can changing the file's access or modification time.

- 2. `stat` command
  - Stat command gives information such as the size of the file, access permissions and the user ID and group ID, birth time access time of the file.

- Example: Create an empty file.


MINGW64:/c/Users/semiyari
create an empty file with the touch
The file is empty
The stat command shows the time of access, born and modified
Modify the file
The time has changed
Use touch again on our file
The file hasn;t changed
But the timestamp has updated

```
semiyari@AU302413 MINGW64 ~
$ touch newfile.txt

semiyari@AU302413 MINGW64 ~
$ cat newfile.txt

semiyari@AU302413 MINGW64 ~
$ stat newfile.txt
  File: newfile.txt
  Size: 0               Blocks: 0          IO Block: 65536  regular empty file
Device: d0ab7935h/3500898613d   Inode: 62487444829859006  Links: 1
Access: (0644/-rw-r--r--)  Uid: (1754655/semiyari)   Gid: (1049089/ UNKNOWN)
Access: 2024-02-02 14:15:50.684161000 -0500
Modify: 2024-02-02 14:15:50.684161000 -0500
Change: 2024-02-02 14:15:50.683547000 -0500
 Birth: 2024-02-02 14:15:50.683547000 -0500

semiyari@AU302413 MINGW64 ~
$ echo "This is the first line" >> newfile.txt

semiyari@AU302413 MINGW64 ~
$ cat newfile.txt
This is the first line

semiyari@AU302413 MINGW64 ~
$ stat newfile.txt
  File: newfile.txt
  Size: 23              Blocks: 1          IO Block: 65536  regular file
Device: d0ab7935h/3500898613d   Inode: 62487444829859006  Links: 1
Access: (0644/-rw-r--r--)  Uid: (1754655/semiyari)   Gid: (1049089/ UNKNOWN)
Access: 2024-02-02 14:17:15.321570000 -0500
Modify: 2024-02-02 14:17:10.366865500 -0500
Change: 2024-02-02 14:17:10.366865500 -0500
 Birth: 2024-02-02 14:15:50.683547000 -0500

semiyari@AU302413 MINGW64 ~
$ touch newfile.txt

semiyari@AU302413 MINGW64 ~
$ cat newfile.txt
This is the first line

semiyari@AU302413 MINGW64 ~
$ stat newfile.txt
  File: newfile.txt
  Size: 23              Blocks: 1          IO Block: 65536  regular file
Device: d0ab7935h/3500898613d   Inode: 62487444829859006  Links: 1
Access: (0644/-rw-r--r--)  Uid: (1754655/semiyari)   Gid: (1049089/ UNKNOWN)
Access: 2024-02-02 14:20:25.542709300 -0500
Modify: 2024-02-02 14:20:20.221857100 -0500
Change: 2024-02-02 14:20:20.221630200 -0500
 Birth: 2024-02-02 14:15:50.683547000 -0500

semiyari@AU302413 MINGW64 ~
$ |
```

- One of the core features of the `touch` command is its ability to modify the access time. To achieve this, we can utilize the "`-a`" option.

- When used by itself, this option will set the access time of the specified files to the current time. If you want to set a specific date, you must use the "`-d`" or "`-t`" options.

EDIT FILES

If we edit a file and type `git status`. We get a message like
Changes not staged for commit:
(SOMETHING WRITTEN HERE)
(SOMETHING WRITTEN HERE)
midified: [name of the file modified in red font]

The process of adding an edited file is exactly same as adding a new file.

Let us modify the file1.txt. Type `echo  world >> file1.txt` where >> means "append" (add as an attachment or supplement). Now let us run `git status`. It shows we have to tracked files (in the staging area) and one modified in working directory file. We have file1 in the staging area but we have modified the file1 and the modified or second version of file is in the working directory. So there are some changes that are not staged yet. We need to run `git` `add file1.txt` now look at the status of our working directory. Both of these files are in the staging area. And we don't have any unstated changes.

COMMIT

- How to commit the snapshots in our stageing area to permanently store in our repository?
  Type `git commit -m "YOUR MESSAGE"`. Where `-m` is for the message. A short message that identifies what this snapshot represents. For instance `git commit -m "Initial commit`. Sometimes a short message is not enough for instance we were working on a bug and we want to explain about something, in this case just type `git commit`. After you enter a VScode will be open.

  Now you find edit the file **COMMIT_EDITMSG** (it is the name of VSCode). On the top you can type a short description (less than 80 character). After that we have a line break (skip a line) and after that we have the long description. Now we save the changes and close the VSCode (by clicking x on top) and back to our terminal and you will see its says the message and how many files has been changes and so on.
  NOTE

- Comments should not be too big or too small.

- We don't want to make commit every time we update a file. In other hand we don't want to code for a couple of days and then make a commit. The whole point of commiting is to record checkpoints as we go. So, if we messed up we always can go back and recover our code. So based on the project your time you spend on your code, you may want to commit. If you work all day, then you may want to commit 5 to 10 times a day.

- As you reach a state that you want to record then make a commit.

- Do not mix the commits. If you are working on a code and fixing a typo and bug. Then you should have two commits one for the bug and one for the typo.

- Use present tense in the commit messages. Instead of saying Fixed the bug, you should say Fix the bug. If you want to use past tense it is fine but make sure you and other team members are use this one.

- **Question** Can we skip the staging area? or do we have to stage our changes before commiting them?
  The answer is no, but you should know what you are doing. Please note that the whole

point of staging area is, if there is changes that need to be reviewed. If you are 100% sure that the changes don't need to be review so you can skip it.

- **Question** How to skip staging area?
  Let us go over it by an example. Type `echo test >> file1.txt`

  - We now skip the `git add` because we are not putting anything in staging area
  - Type `git commit -a -m "Right Your Message here"` where `-a` is for all.
  - We also can combine both `-a` and `-m` together and write `git commit -am "Type Your message"`
  - If you look at the message it will tell you how many files was changed and how many insertion we had. If we have lines deleted then we had deletion

  What are "insertion" and "deletion"?
  Insertion is the number of lines inserted and deletion is the number of line deleted.

HEAD POINTER

- *Head* is a reference variable. We call this variable a pointer because its purpose is to reference or point to specific commands to a repository. As you make new comment the pointer changes or move to a new comment.

- The *Head* always point to the current branch of our repository. Imagine we have 3 versions of a file with commit. Head will point to the last commits that we make. It is the *parent* of the next commit that we make.
- Head becomes more important when we start talking about *branches*. By default the branches that we are working is called *master* (or sometimes *main*) branch.
- If we make a new branch then header moves to commit on this branch.
- We have ability to move Head back and forth to different branches, so we can write on different branches.
- Git will keep track of moving the Head around for you.
- Remember inside of our project folder we have a directory called `.git` and inside of `.git` we have *Head*
  - Type `ls -la .git`
  - To see what is inside of file *Head*, type `cat .git/HEAD`. It returns something like "ref:refs/heads/master"

  - Type `git log`
    It will provides all changes, commits, authors, and more. You also see that the pointer (HEAD) is pointing the SHA value to the master branch.

- **How to get the "SHA Value" directly?**

```
git rev-parse HEAD
```

REMOVE FILE

Let us say that we do not need the "file2.txt" any longer. We use `rm` command to remove the file. This command is a **UNIX** command.
- Type `rm file2.txt`

- Let us now run the `git status`. It shows that we have one change that is not stage for commit. (Not in staging area). So the file is removed from working directory but still is in staging area. To see this type
  - `git ls-files`. These are the files in staging area and you will see that the both files are there.
  - Remember whenever we make a changes to a file we have to stage the changes by `git add` command. Now we have to do that here.
  - Type again `git add file2.txt` Now we have report what we have done in working directory and updated the stage area.
  - Type `git ls-files` and you will see the "file2.txt" is not in the staging area any longer.
  - Type `git status` you will see you have one change ready to be committed and indicated with green font "deleted file2.txt" that means the change is in the staging area and ready to be committed.
  - Type `git commit -m "Remove unused code"`. Now you see in terminal that one file is changed and one is deleted.

  Question:
  Is there any way we can do both these command only with one command?
  Instead of using `rm FILE_NAME.txt` and `git add FILENAME.txt` to remove the unwanted file from both directory and staging area. We can use `git rm` command instead of `rm` UNIX command, so if you type

  - `git rm FILENAME.txt` or if you have multiple files you can type
  - `git rm FILENAME1.txt FILENAME2.txt` or you can use a pattern
  - `git rm *.txt` or whatever the extension of the file is!

RENAMING or MOVING FILES

- Currently we have "file1.txt. in our working directory. We want to rename to"main.js". We will use the `mv` command (move from UNIX) to rename file1.txt.

  - Type `mv file1.txt main.js`. Then type `git status` Now we see we have to changes and both of them are unstaged because they are indicated by red font. We have deleted operation "deleted: file1.txt" and we have a new untracked file "main.js".
  - As you know by now, git doesn't automatically track all your files everytime you have a new file in your project. You have to add it to staging area so git starts tracking it. So, we have to use `git add` command to stage both these changes.
  - Type `git add file1.txt` and `git main.js` or you can type `git add file1.txt main.js`
  - Now you need to type `git status`. You will see both files are green which means both are in staging area. Also git recognize that we have renamed file1.txt to main.js. If you look at the terminal you will see "renamed: file1.txt -> main.js"
  - So renaming or moving files is a two-step operations. First we have to rename or remove the file from working directory and then we need to stage two types of changes addition and deletion. So, similar to *removing* file, git gives us special command to do both step in one. It is `git mv`. Note 'git mv OLD-File NEW-File.
  - Type `git mv main.js file1.js`. Now if you type `git status` you will see the change has applied to both working directory and staging area. Now let us commit the changes type `git commit -m "Refactor code"` Look at the *statistics* one file was change with 0 insertion (we didn't add any new lines to any files) and 0 deletion (we haven't remove any lines from any files)

IGNORING FILES

Almost in every project, we have to tell git to ignore a certain files and directories. For instance, we don't want to include log files or binary files. These files are generates as a result of **compiling** our code. Adding these files just increases the size of our repository without adding any valuable information. Let us create a new directory called **logs**. - Create **logs** directory. - Type `mkdir logs` and then add a log file here, by - `echo hello > logs/dev.log`. Now let us get the status - `git status`. Now git saying that we have an untracked directory called logs. **But we don't want to add this into staging area** Because we don't want to git to track this. So to prevent this we have to create a special file called `.gitignore`.

- Create **`.gitignore`**. This fies has no name just has an extension. It should be in root of your project. So, type
  - `echo logs/ > .gitignore`. Now we need to open file using VSCode/
  - Type `code .gitignore` So in this file we have a single entry “logs/”. Which indicates a directory. We can list as many files and directory as we want here for example we can include “main.log”. We can also use pattern like `*log` (all log files).
    - In the VSCode underneath of “logs/” type “main.log” and underneath of this one you can write “*.log”
  - Once we are done we save the changes and back to terminal. Now run
  - `git status`. You will see that git no longer says that we have a new directory called logs because it is **ignoring it**. Instead it says, we have a **new file** and it called **.gitignore**. Now let’s add this file to staging area and commit our code
  - Type `git add .gitignore` and `git commit -m "Adding gitignore"`.
- So this is how git ignores a file or directory. But remember, it works if you have not already included a file or directory in you repository. If you have files added before creating `.gitignore`, those file will stay in your repository.
- Let me show you what happen in this case by an example. Let us create a new directory called **bin**. Imagine this directory has our compile source code so using the echo command `bin/app.bin`. And we are going to accidently include this to our repository before creat the ignore file.
  - `mkdir bin`
  - `echo hello > bin/app.bin`
  - `git status` Now we have anew directory and we want accidentally commit it to our repository
  - `git add .` Remember `.` means add all the changes
  - `git commit -m "Add bin."`. Here is one problem, every time we **compile** our code git is going to say that the file “bin.app.bin” is changed. (Remember any code your compile its bin will change). Thus we have to stage and commit it.
  - Back to file “.gitignore” we need to **add “bin/”** save it and back to terminal.
  - How to go to “.gitignore” file? You can directly open it or type code
    `code .gitignore` on terminal.

- `git status`, you will see you have modified ".gitignore". (remember we added "bin/"). Now let us stage and commit it.
- `git add .`
- `git commit -m "Include bin/ in gitignore."`. This time git is not going to ignore changes in this directory. Because it's already tracking this directory. So Let us modify our bin file by saying
- `echo hello world > bin/app.bin` look at status
- `git status`. Git says the file is modified. This is not what we want. We want this file to be ignored. Now we want to remove this file from staging area. Which we are what proposing for the next commit.
- The command `git ls-files` shows the files in staging area. As you see the **bin** file (or directory) is in staging area. We should remove it. With remove command (`git rm`) we can remove a file from both working directory and staging area. But in this case **we don't want to remove it from working directory**. Why? because this is how we launch our application. So we want to remove it **only from staging area.** How? By using "options" for `git rm [option]` command.
  * Option: `--cached` We can remove only from the index (or staging area. Index is the old term for staging area).

  * Option: `-r` allow recursive removal

  (You can find more help about `git rm` command by typing `git rm -h`, where `-h` means for help.)
  * `` `git rm --cached bin/` ``

    If you just type only option `--cached`. You will get error " fatal: not removing 'bin/' recursively without -r". Thus you need to type
    `git rm --cached -r bin/`. Now the entire directory will be removed from staging area. To verify just type
  * `git ls-files` and you will say the directory bin is not there.
  * `git status` We have one change that is ready to commit. The directory is deleted from staging area.
  * `git commit -m "Remove the bin directory that accidentally was committed"` From now on git will not track changes from this directory. So if we type
  * `echo test > bin/app.bin` and then `git status` we see nothing to commit and working tree is clean.

  If you go to *github.com/github/gitignore* you can see various "gitignore" template for different programming lnguage. For R is the file "R.gitignore".

SHORT STATUS

The command `git status -s` provides you with a short status. The modified file will be shown by red font capital M and untracked by ?? in red font. For instance if you run `git status` and you have "Modified: file1.js" and "untracked files: file2.js". If you run `git status -s` you will get "M files1.js" and "?? file2.js" The green A means the file is added and green M means the modified file is added. - You may want to look at `git status -h`. For instance you will see that `git status -b` gives you information about branch.

- Let us have an example.We are going to modify one file and add a new file
  - `echo sky >> file1.js` We are appending sky to the file. Let's create anew file
  - `echo sky >  file2.js`. Now let us run the status
  - `git status`, it shows we have one modified file and one new file. The outout is very comperhancive but very wordy.
  - `git status -s`. It looks much better. We have two columns. The **left column** represent the **staging area** and the **right column** represents the **working directory**.
    - We have modified file1, we have some changes but this changes are not in the staging area. It is in the working directory and we don't have it in staging area. That is why the left column is empty and we have a red M in the second column,
    - (Second row) we have two red question marks in both columns.
  - Now let add file1 to staging area `git add file1.js`. Now run another short status
    - `git status -s`. Now for file1 we have a green M in the left column or in the staging area column. On the right column we don't have anything. Why? Because we don't have any extra changes. What happen if we modified file1 another time?
    - `echo ocean >> file1.js` and the run the short status `git status -s`. Now in the lef column we green M, which means we have some changes in the staging area, We also have a red M in the working directory column which indicates we have some changes in the working directory tha are not added to the staging area. Now let us
    - `git add file1.js` and then `git status -s`. Now, we haveonly one green M in the staging area column, It means all changes we have in working directory are in the staging area.
  - Now let us add file2 to staging area and then run short status
    - `git add file2.js` and then `git status -s`. Now we see that in staging column we have Green M for file1 which indicated is modified and we have Green A for file2 which indicates it is added. If we delete a file it will be denoted by **D**.

VIEWING STAGED AND UNSTAGED CHANGES

Before we committing what we have in the staging area we need to **review our code** because we don't want to commit bad or broken code to our repository.

- **Question** the `status` command shows that the files that have been affected. How can we see the exact lines of code that we have staged? We need to use `diff` command.
  - `git diff --staged`. Now we see what we have in staging area. Comparing files using terminal widow is not really the best wey to do it. It is better to use *Visual Tools*. Let us now explain the output of this command and then we show how to do it by a visual tool.
  - At the top you see the diff utility was called. Something like *diff –git a/file1.js b/file1.js* So we are comparing two copiestwo copies of the same file. The first copy *a/file1.js* is the old one. Which we had in the last commit. and the second copy *b/file1.js* is what we currently have in the staging area.
  - Below that we have *index ...*, which is some metadata and it doesn't matter.
  - Next we have the "legend". So changes in the old copy are indicated by a **Minus (-)** sign and changes in the new copy indicated by **plus (+)** sign. -Now we have "header" with some information about what part of our file has changed. Our file is very short and we have only a few lines of text. In reality the files can have hundred lines of code. In this case git divide them into chunks and every chunk has header with some information. Here we have only one chunk and one header. We have two segments
    - The first segment is *prefix with a minus sign* gives us information about old copy and the second segment is a *prefix with a + sign* this contains information about new copy, which is what we have in staging area. For instance the header is
      *@@ -1,3 +1,5 @@**
      hello
      world
      test
      +sky
      +ocean*

      - So old copy start from 1 and 3 lines have been extracted *-1,3*
        hello
        world
        test

      - New copy start from 1 and 5 lines have been extracted So old copy start from 1 and 3 lines have been extracted *+1,5*