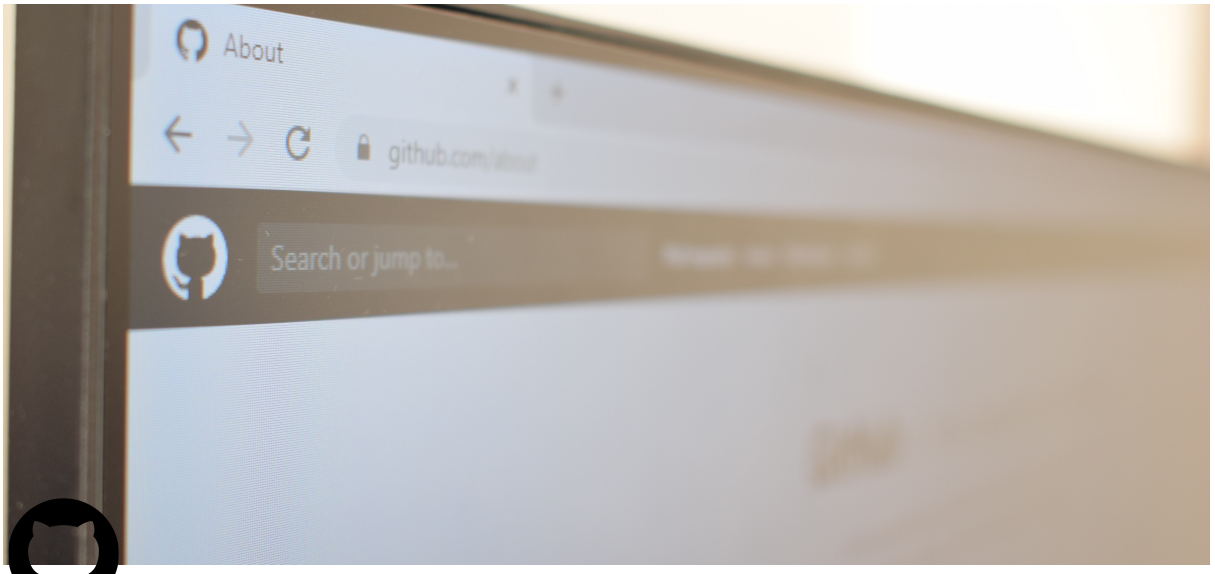


Git & GitHub

> **Author:** Tarek Hasan

> **Info:** Git and GitHub have transformed the landscape of software development. Git, a distributed version control system, empowers developers with features like branching and merging, enabling efficient collaboration and code management. GitHub, a web-based platform built on Git, enhances collaboration by hosting repositories and offering tools such as pull requests and issue tracking. Together, they streamline the development process, providing a solid foundation for version control and fostering social coding practices.

📌 In this whole section we are going to learn what is git and GitHub, what are their usage and purposes. We'll make ourselves familiar with their functionality. Meanwhile we will gain knowledge of developing a project using git and GitHub.

▼ Topic List

▼ Version Control System [VCS]

▶️ **What is version control system?**
A version control system is like a management system. It manages the changes in a program, code or website or anything like that. Suppose you've an website which was built 10 months ago. You have committed many changes to the website. What version control does is that it creates a new version of the old code/program every time you make changes. Suppose your website is in version 10. You added a new page to the website. Now you website is in the version 11 or 10.1 however you may define it. It controls the version of your program/code every time you make changes to it

▼ What is the purpose of version control system?

▶️ As it creates previous version of the program/code we can easily access the previous version. It's like a backup system. We can also rollback to the previous version. Suppose you are working on a big project within a team. So the team has 10 members and everyone can access the project source code. Everyone can make changes. With the help of version control system you can see who made changes, when did someone made changes and what did they change. So you can say it is a tracker or it works as a tracker.

📌 **There are 3 types of version control system:**
- Local VCS
- Centralized VCS
- Distributed VCS

▼ Local VCS

- Changes are stored in a database with timestamps.
- Code is in local system.

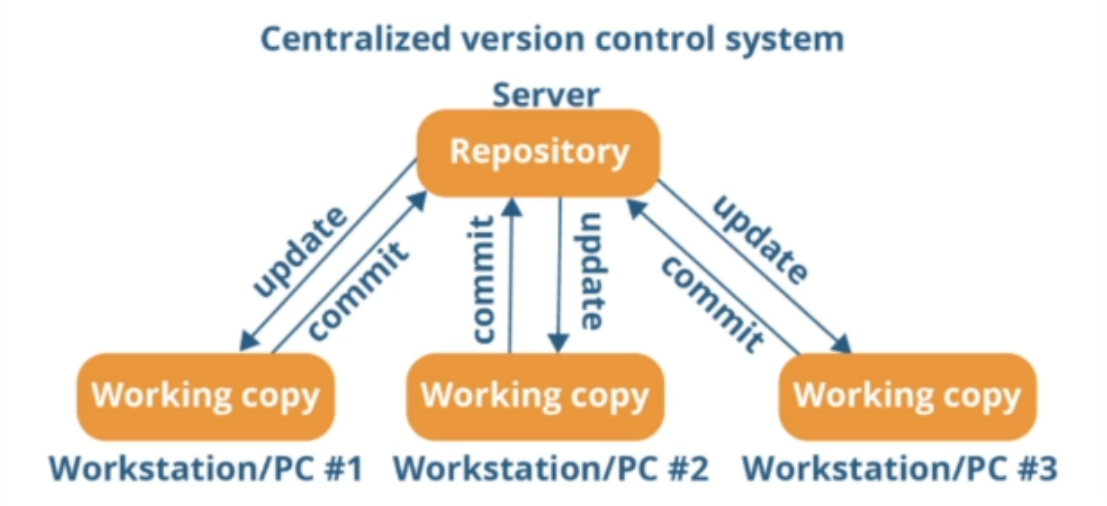
> ⭐ **Cons:** There is a high risk of loosing the project like if your hard disk fails or gets corrupted you will loose the project as it is stored in local hard disk and has no backup.

▼ Centralized VCS

To know about centralized VCS we first need to know about repository. **So, What is a repository?** Well a repository is as same as folder in you computer. There are 2 types of repository:

- **Local repository** - [Folder in you computer or laptop]
- **Central repository** - [A folder hosted in google server, AWS or Git-hub can be called central repository]

Centralized VCS is based on the central repository system. Now look at the picture below:



As you can see in the centralized VCS there is only one repository. This repository can be hosted in any web server like AWS, Git-hub etc. Suppose 3 users are working in a project which is under centralized VCS. So it has only one repository and users are using the repository all together. They are editing their code and committing update to the repository. But no user has the local copy of the repository. They have the local copy of the file which they are editing and

committing to the repository hosted on the web server but they don't have the repository locally. As there are no local copy, it has some cons.

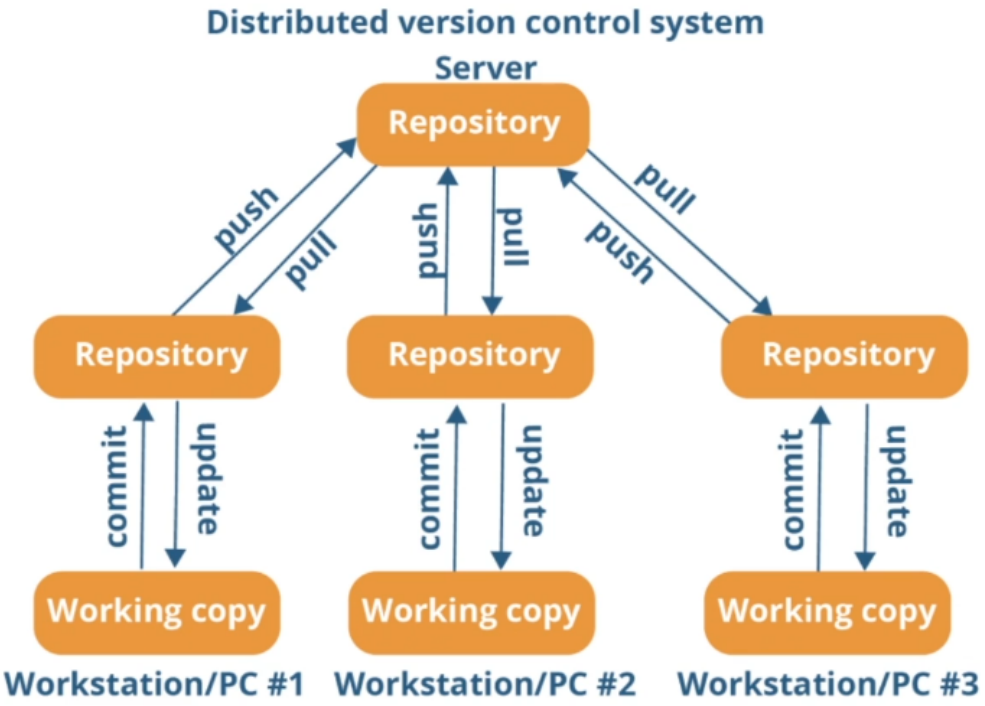
> ⭐ **Cons:**
> 1. If the central repo goes down for an hour or the we server goes down for and hour no user will be able to access the repository or commit changes.
> 2. If the hard disk of storage of the web server goes corrupt or down, users will loose the project.

▼ **Distributed VCS**

> ▶️ Distributed version control system is the most used and most popular VCS.

Distributed VCS contains multiple repositories. How? In distributed VCS each user has their own local repository. They have their own files in their own local repository. There is one central web-hosted repository. It is the final repository where the main code/project resides. So the main code stays in central repository. A user can pull the code from the central repository to his own local repository and edit the code and commit changes by pushing the edited code again in the central repository. Suppose you are working in a project and there are another 100 different programmer who are also working on the project. Everyone has the whole code in their local repository. When someone does something on their local code they then pushes the code to central repository which updates the old code which is in the central repository. Every time someone pushes the code and makes changes it creates a new version of the old code of the central repository so that the previous code stays the same and anyone can access the previous code.

> ⭐ Pros:
> 1. It provides full backup. If the server stops working or crashes, users already has their code or full project in local repository.



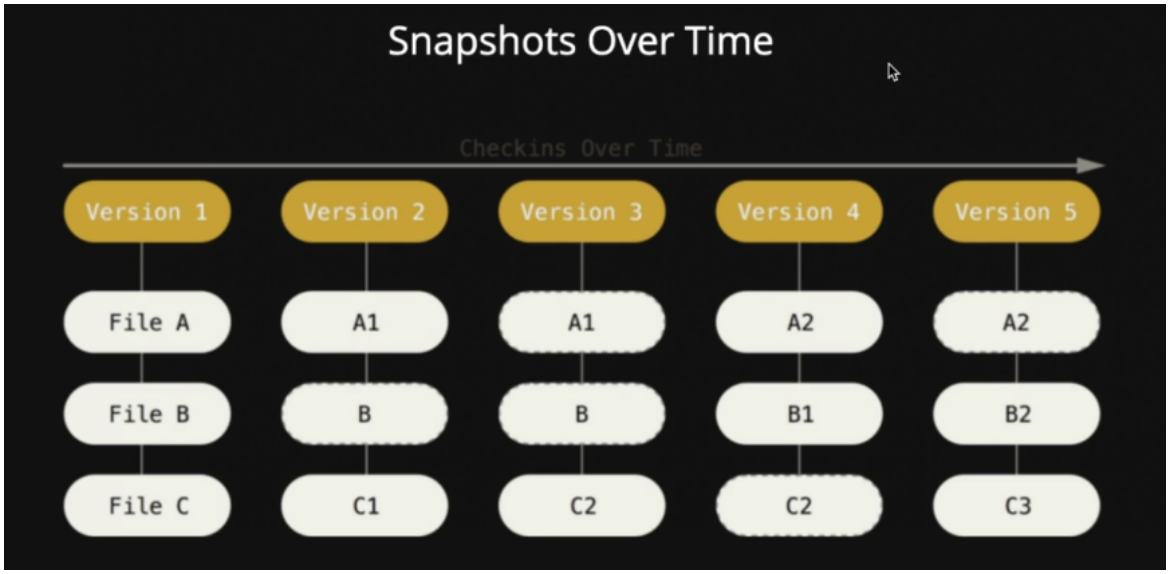
▼ How was Git created?

> 📌 Git was created in 2005 by Linus Torvalds who is also the creator of Linus kernel.

Linus Torvalds started working on Linux in 1991. Within 2002 there were several developers behind the Linux kernel and the source code was so big. At that time, Linus Torvalds was using a VCS system called BitKeeper. But it wasn't fully meeting his requirements. He needed speed, more secure system. That's why he created Git.

▼ What is Git?

▶️ Git is a free open source distributed version control system designed to handle everything from large to small project with speed and efficiency. It stores snapshot of a project but not differences.



Suppose you're now in version 5 of your project. Now you want to see what it was like in the beginning. You can simply rollback to the version 1 and see the code.

▼ Features of Git

- Distributed
- Almost everything is local
- Non-linear/Branching so that the code doesn't break
- Secure
- Speed

▼ Git Installation

▼ Mac

▶️ If you don't have homebrew installed on your Mac install it by running this command:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.
```

▶️ After doing that, install Git by running this command:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.
```

▶️ To check the version of Git run this command:

```
git --version
```

▼ Windows

▶️ Go to this website:

> Git - Downloads
>
> The entire Pro Git book written
> by Scott Chacon and Ben Straub is available to read online for free.
> Dead tree versions are available on Amazon.com.
>
> https://git-scm.com/downloads

Then download the .exe file for windows and install it. Use Git-bash to operate git.

▼ Git Configuration

▶️ What does it mean by Git configuration? When you're working on a big project, there must be a lot of contributors who are working at the same time. Whenever you change something to the project and push the code git should understand who changed the code in the latest version. That is why we need to configure git. How are we supposed to do that? Let's see:

```
git config --global user.name "NAME"
git config --global user.email "EMAIL ADDRESS"
```

To check the configuration you can use this command:

```
git config --list
```

▼ Some Commands

▶️ You may already by familiar with command line tools like the terminal in mac and linux. But windows doesn't have it or you can't use cmd or powershell as command line tool like terminal of mac and linux. To operate Git we can use Git bash which is already installed in your application list if you've installed git.

▼ Commands

> 📌 pwd - Prints the directory you're in

▼ ls command

▶️ ls command shows you the content of the directory you are in. But this command has some functionality:

📌 ls [shows the content of the directory you're currently in]

📌 ls -l [shows the content in a list format]

📌 ls -a [shows all the content including hidden contents]

📌 ls -al [shows all the content including hidden contents in a list format]

▼ Changing directory

▶️ To change directory we can use cd command. Suppose in the directory you are currently in you have a folder called Desktop. You can access Desktop folder by changing the directory with cd Desktop command. You can go back to your previous directory using cd .. command.

▼ Creating directory and files

▼ Directory opening and deleting

▶️ Suppose you're in the directory called Desktop. Now you want to create a folder named FOLDER1 in you Desktop. We can do this by mkdir command.

📌 mkdir FOLDER1

This will create a folder named FOLDER1 in your Desktop. Now you want to create 2 more folders named FOLDER2 and FOLDER3.

📌 mkdir FOLDER2 FOLDER3

This will create 2 folders in a single command. Now suppose you want to create a folder named FOLDER4 in the FOLDER3.

📌 mkdir -p FOLDER3/FOLDER4

This will create FOLDER4 inside the FOLDER3. Now, to delete a directory we can use the rmdir command. Suppose we want to delete the directory named FOLDER2.

📌 rmdir FOLDER2

This will remove the directory. If you have something inside the folder I mean some contents then you can delete the directory along with those contents using this command

📌 rmdir -R FOLDER2

▼ **File opening, accessing and removing**

📌 touch HELLO.txt

This will create a text file in the directory you are currently in named HELLO. You can create any file or any type of file using this command. To access the file we can use:

📌 open HELLO.txt [For mac]

📌 start HELLO.txt [For Windows]

To delete the file we can use rm command.

📌 rm HELLO.txt

This will remove the file.

▼ **Copy, Cut and Paste**

▼ Copy paste

▶️ Suppose you are in a directory which has a folder named FOLDER1 and a text file named HELLO.txt. Now you want to copy the HELLO.txt file in the FOLDER1.

📌 cp HELLO.txt FOLDER1/HELLO.txt

By running this command we can copy the file in the FOLDER1. The structure of the command is

📌 cp [FILE NAME] [FOLDER NAME]/[FILE NAME]

You can also rename the file while doing is like this:

📌 cp HELLO.txt FOLDER1/HEHE.txt

It will rename the file while doing the copy paste.

Now you have a file named HELLO.txt in the directory named FOLDER1. You can now copy the file from that directory to your current directory by running:

📌 cp FOLDER1/HELLO.txt HELLO.txt

Now suppose you want to copy all the content of the directory FOLDER1 to a new directory named FOLDER2. To do that we can do this:

📌 cp -R FOLDER1 FOLDER2

This will copy all the contents from the FOLDER1 directory to newly created FOLDER2.

▼ Move paste

Basically you can just change the cp command to mv and it will move the content to wherever you want. This is an example:

📌 mv HELLO.txt FOLDER1/HELLO.txt

It will move HELLO.txt to the directory named FOLDER1.

▼ Introduction to Git

▼ Types of Git commands

▶️ Creating Repository [Local]:

📌 git init

▶️ Making Changes:

📌 git add
git status
git commit ____

▶️ Parallel Development:

📌 Branching:
git branch
git marge

▶️ Syncing Repository

📌 origin
remote
pull
push
fetch

▼ Three State Architecture

▶️ This means three states. There are three states which are:

📌 - Modified (Working Directory)
- Staged (Staging Area)
- Committed (.git Directory(Repository))

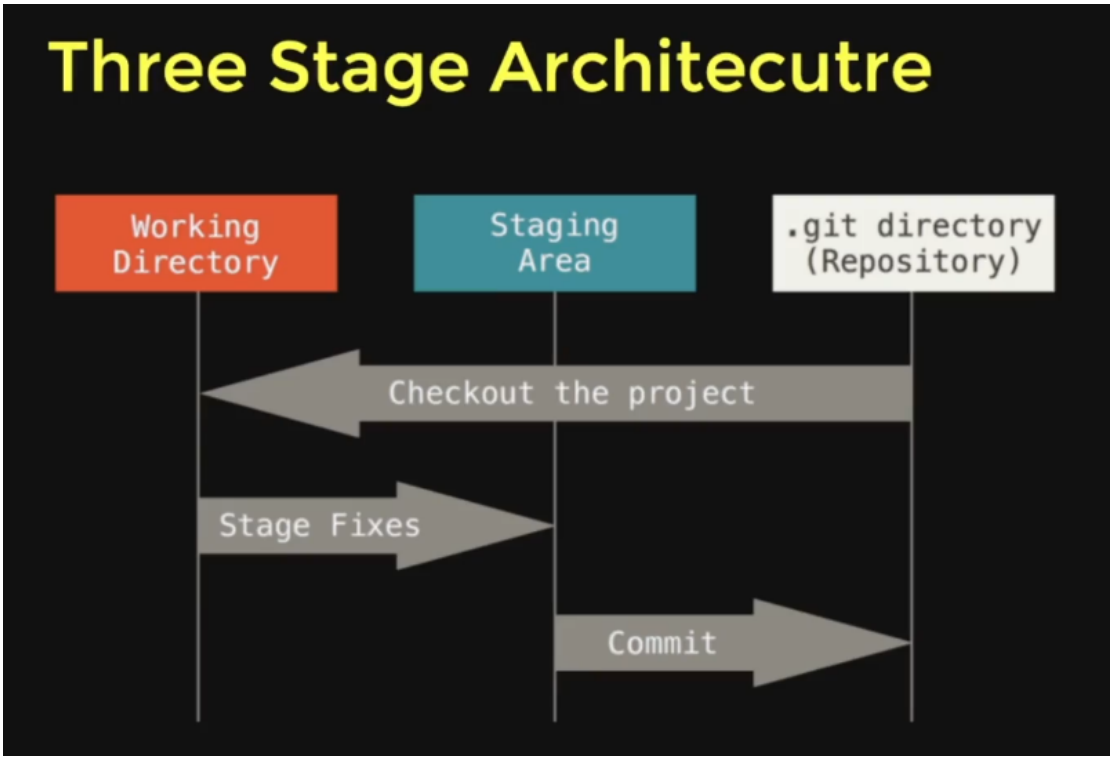
▶️ What are these? Let's explain:

⭐ **Modified (Working Directory):**

Modified means the working directory. Suppose all your project files are in a directory named Project1. This project1 is the working directory. It is the local directory which is on you computer. It is not the directory hosted in a server. It is the directory where you store those project files and constantly modify.

⭐ **Staged (Staging Area):**
So suppose you have done editing the file. Now you want to commit the files to the git repository. But there is a preparation stage. This is the stage. Like after modifying your files you want to commit those files. This is the V1 of your files. Now you need to make it clear that what files or modification you want to save as version one. This is what staging files means.

⭐ **Committed (.git Directory(Repository)):**
Now what does committing means? Committing means saving in this case. After you're done editing you obviously will save right? There will be a .git directory(Repository) in the project folder if you initialized git. What committing will do is that it will take a snapshot of the current state of the files. Then it will save them as version 1. Then when you make some more changes, it will again save them as version 2 while already keeping the version 1. That's how you can access the previous version.



▼ Initializing Git Repository

Initializing/Creating a git repository has 2 types such as:

- **Initializing you own local repository**
- **Cloning a remote repository**

> ⭐ **Initializing you own local repository:**
>
> This means creating a whole new git repository from scratch.

> ⭐ **Cloning a remote repository:**
>
> Suppose you have a website already which is well build. Now your friend want to contribute to the development of that website. He can do that by cloning the repository of yours which already has all the source code. He doesn't need to create a new repository.

▼ Creating a git repository

▶️ To create a git repository we can use the below command. This will create a folder named .git which is also a hidden folder inside our project folder. It is the folder where all the snapshot will be stored.

⭐ git init

Now let’s get on with the practical implementation. Suppose you have your project folder named “My_Project”. Inside the folder you have your file which is “Index.html”. There is a command which checks your git status. It will show you if you have git initialized in the project folder or not.

⭐ git status

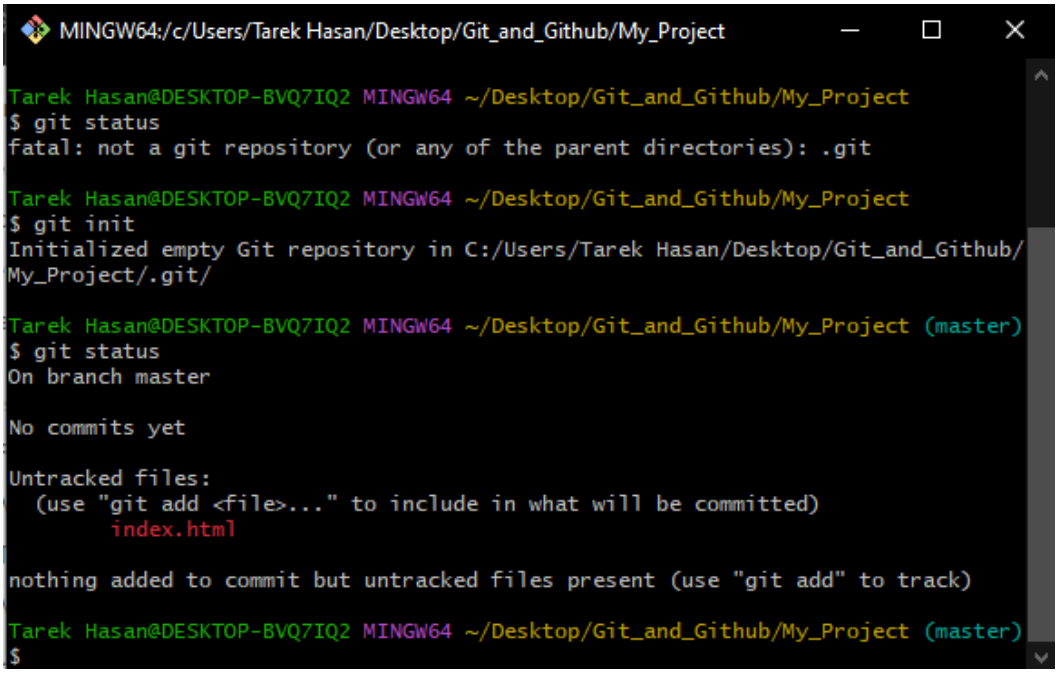
If you don’t have git initialized in the project folder then the command will show an output like this:

📌 fatal: not a git repository (or any of the parent directories): .git

After running the “git init” command it will show and output like this:

📌 Initialized empty Git repository in C:/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project/.git/

After running the “git init” command git will take over the project. You will be in the master branch then. After running the “git status” command again you will get to see something like this:



But look at the status. Git says the file is untracked. Because the file “index.html” is still in the modified stage. First we need to get it into the staging stage and then we need to commit the file. Also there is no version to track yet.

▼ Tracking Your Project

▼ Tracking Files

▶️ Each file in a project/working directory has 2 states:

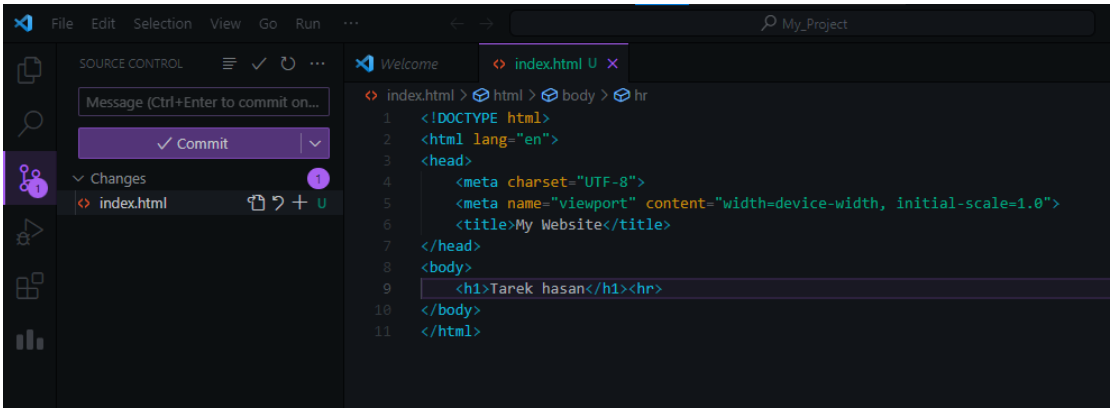
- Tracked
- Untracked

What does these means?

> ⭐ **Tracked:**
> Tracked files are those files that git knows about. Means you have committed those files and there is a snapshot already saved

> ⭐ **Untracked:**
> Any files in the working directory which hasn’t been committed yet or which doesn’t already have a snapshot is untracked file.

Now as we already know we are working with the “index.html” file inside the project directory. We will be using VS Code which has built in git implementation.



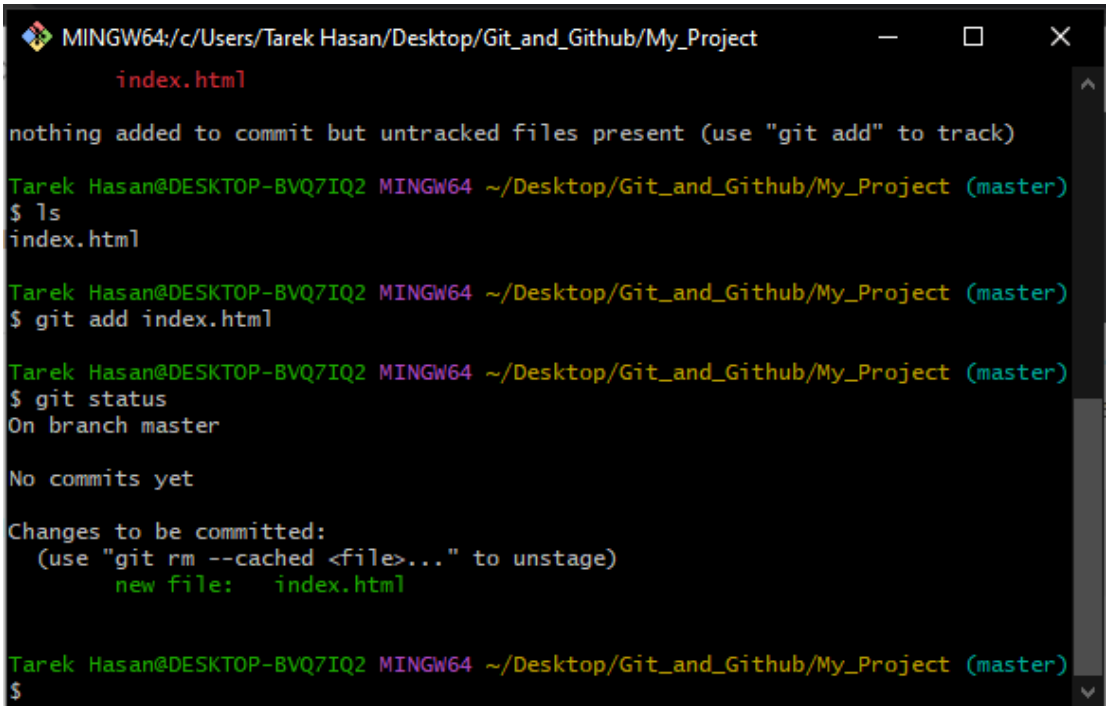
This is the option which is related to git. Now the “U” beside the file tells us that the file is untracked. Now we need to take the file into staging state. To add the file into staging area we can use this command:

> 📌 git add <FILENAME>

It will only add the file which we want to. To add all the file in the project directory we can use this command:

> 📌 git add .

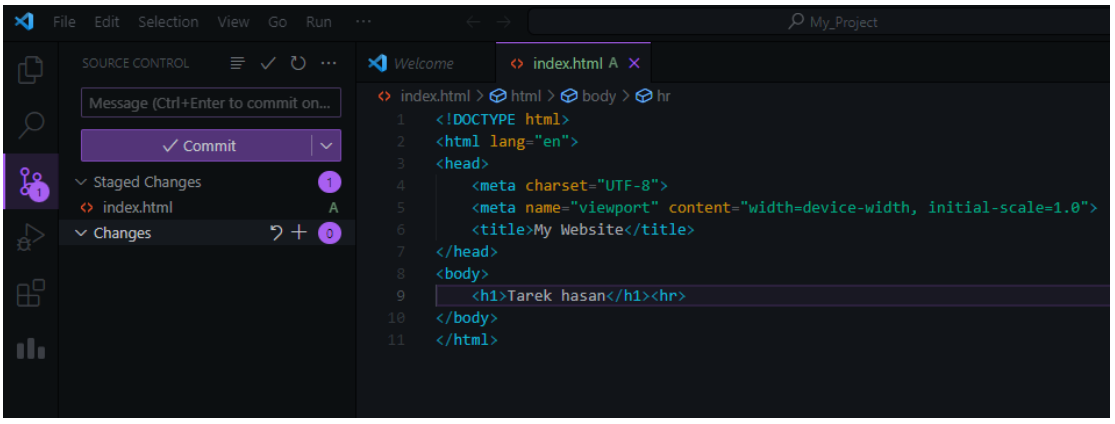
After running the command out git status will be like this:



Look at the picture carefully. It shows that we are on the master branch, we didn't commit yet. But the "untracked" keyword is now missing. Cause it is now tracked. and it says "new file: index.html". It also shows us the command to unstage the file as it's already on stages state. To do that we can use:

> 📌 git rm —cached <FILENAME>…

But git doesn't know about the file. We need to commit the file now. But let's see what has been changed in the VS Code.



We can confirm that the file is in staging state.

▼ Commit Changes

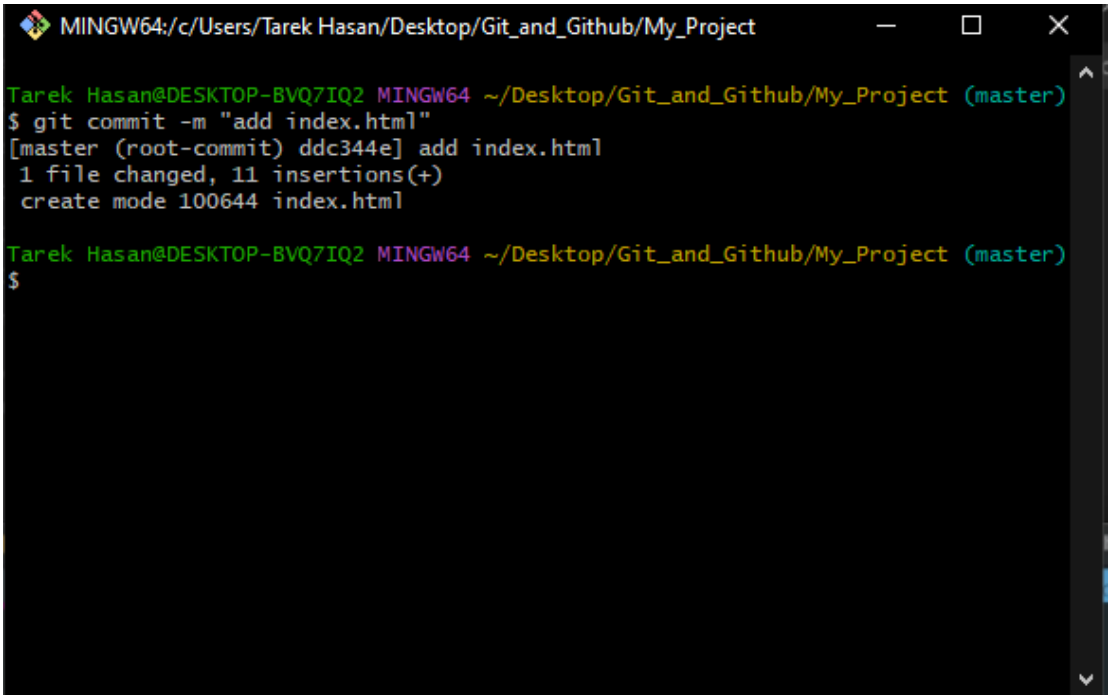
▶️ What is commit? Moving files from staged area to git folder is called commit. Making commits, creates a version/snapshot of your project/project file.

To commit the project we can run this command:

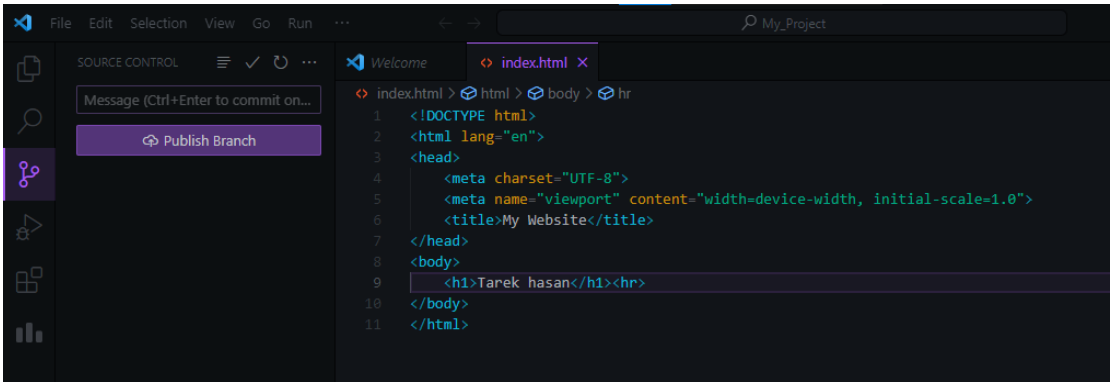
> 📌 **git commit -m "Add your message here"**

▶️ Note: Here the "-m" is a flag. Anything we write using "-" is considered a flag and it has it's own functionality. "-m" flag means message. We can add any message in here.

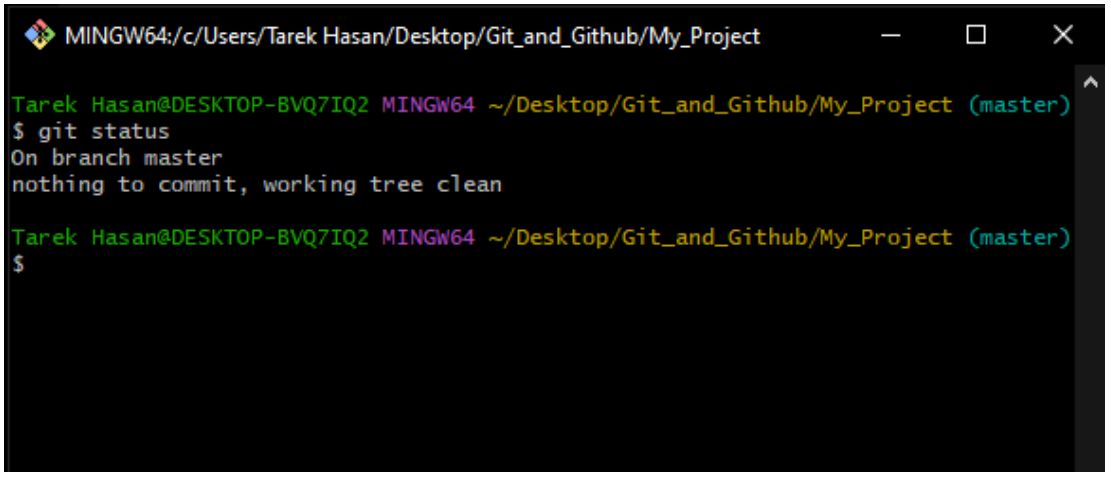
After running the command it will give us a output like this:



This means we have committed the project. Let's see what is the condition of VS Code:



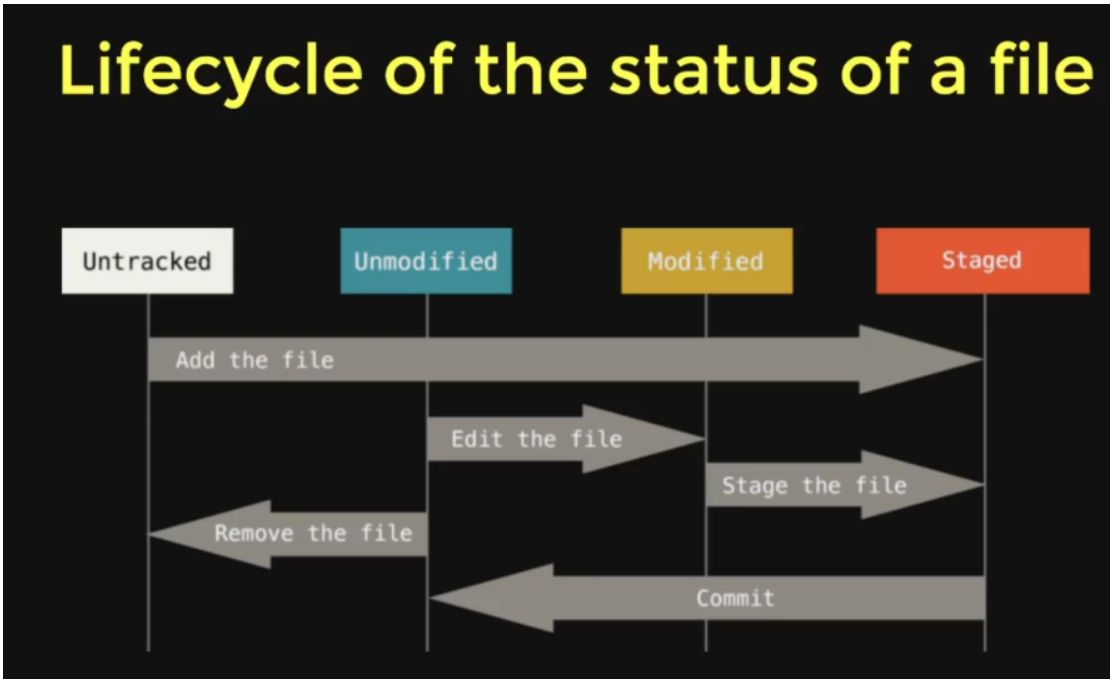
As you can see there is nothing in the side panel right now. That means we have committed the file and now we have created our first version/snapshot. Let's see what is our git status now:



It says nothing to commit.

▼ **Lifecycle of The Status of a File**

▶️ Look at this picture:

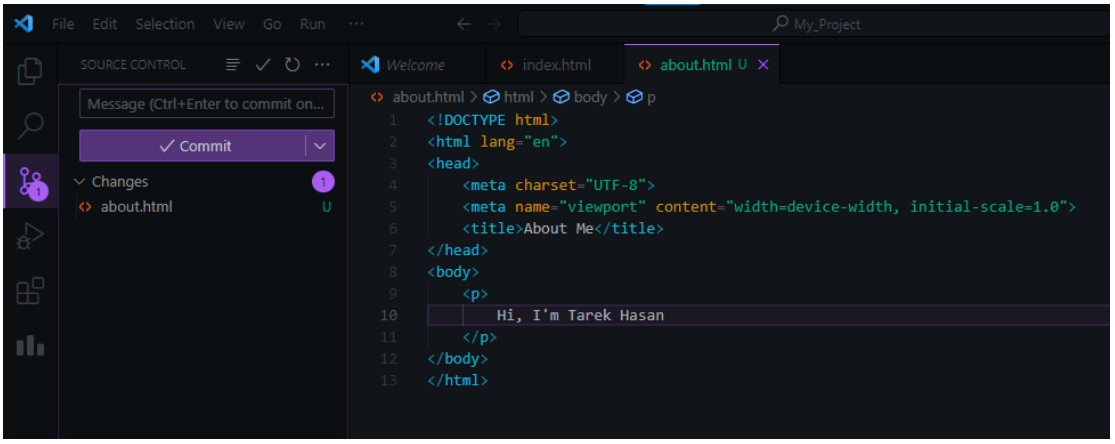


Look at this picture carefully. It shows how the files cycles back and fourth in these steps when we stage a file or commit a file. Let’s explain the picture:

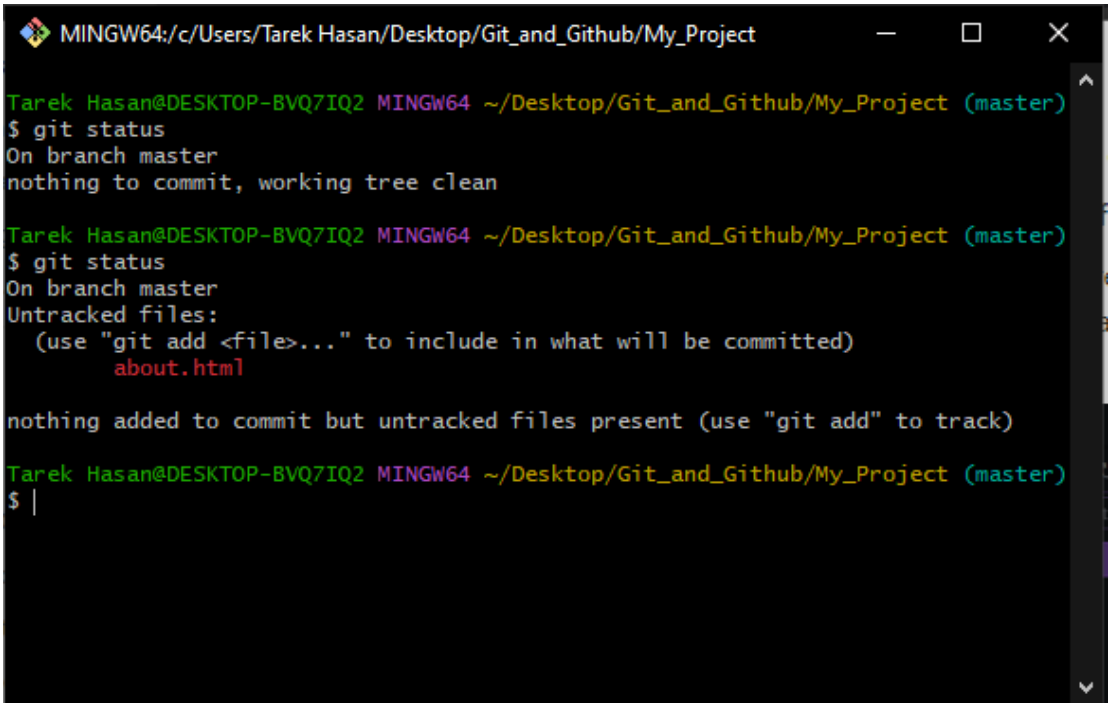
⭐ We have already committed out “index.html” file right. Suppose we have added another file named “about.html”. What will be the status of this file? It will be untracked cause we haven’t staged or committed this file. But let’s talk about out “index.html” file. What’s the current status of this file? It is now in “Unmodified” state. Cause we have committed the file but haven’t changed anything inside the file after that. If we change anything in “index.html” then it will go to the “Modified” state. Then if we stage the file using “git add <FILENAME>” command it will go to the staged state. Then we can commit the file. But what about our file named “about.html”? It will not go to the Unmodified or Modified state. Cause we haven’t committed yet. We have a version of “index.html” which is unmodified. But we don’t have a version of “about.html” yet. So we will have the stage the file. Then we can commit and then it will go to the “Unmodified” state.

▼ Example of Lifecycle of The Status of a File

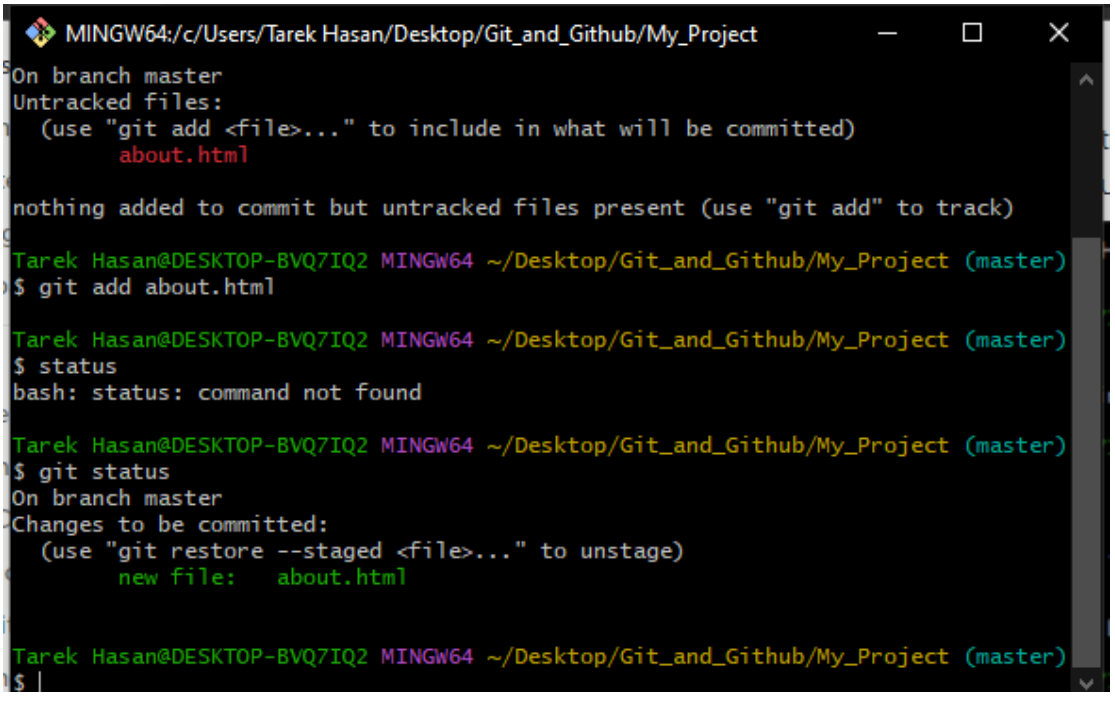
▶️ In this section we are going to look at the practical example of the lifecycle. We are going to create a file named “about.html”. The look at the VS Code status and check the git status:



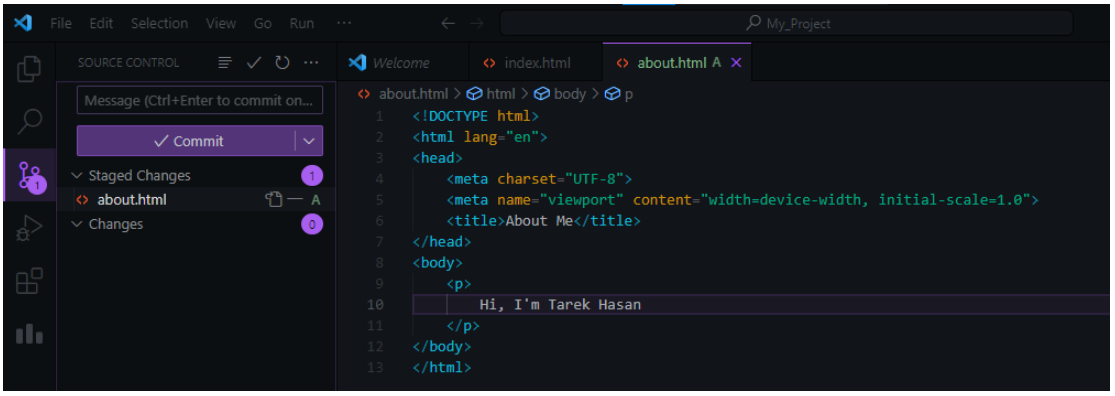
Look at the VS Code after creating “about.html”. It says Untracked as same as before. Now let’s look at the git status:



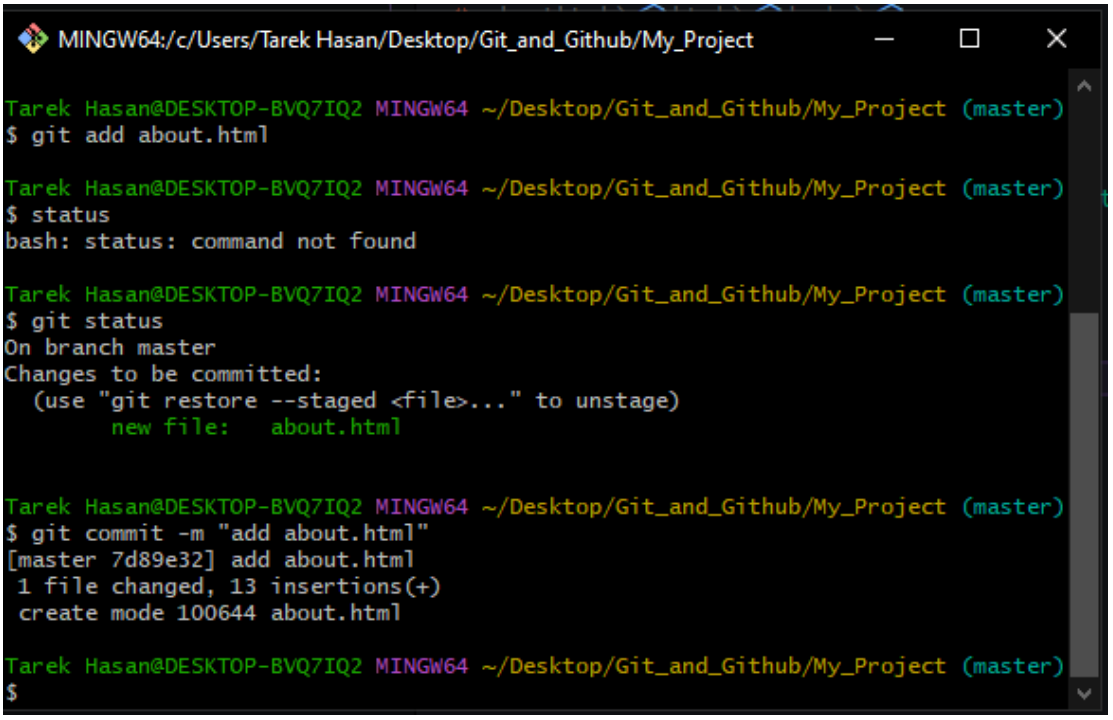
It also shows that “about.html” is untracked. Now we are going to take the file into staging area.



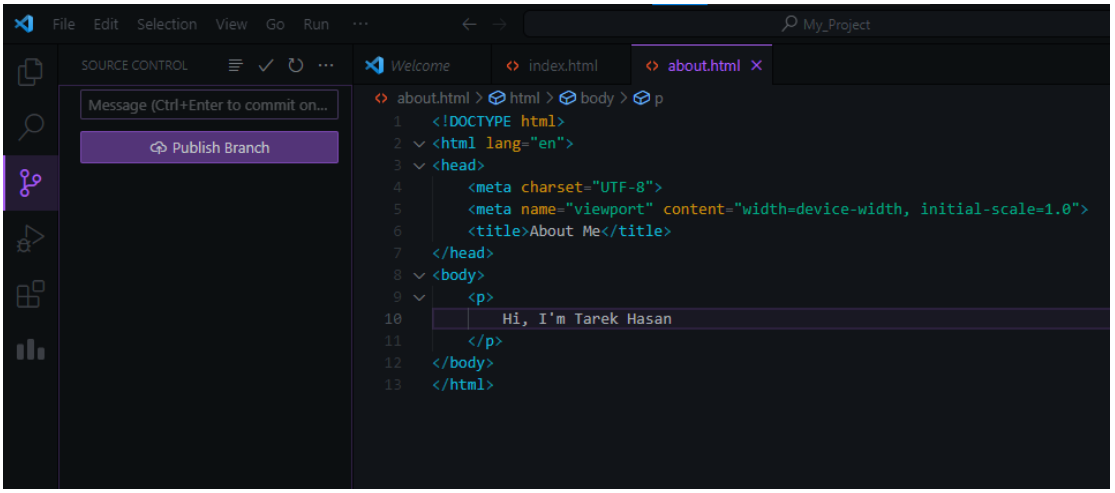
After doing this the git status command shows that the file is in the staging area but is not committed yet. Now let’s see the VS Code state:



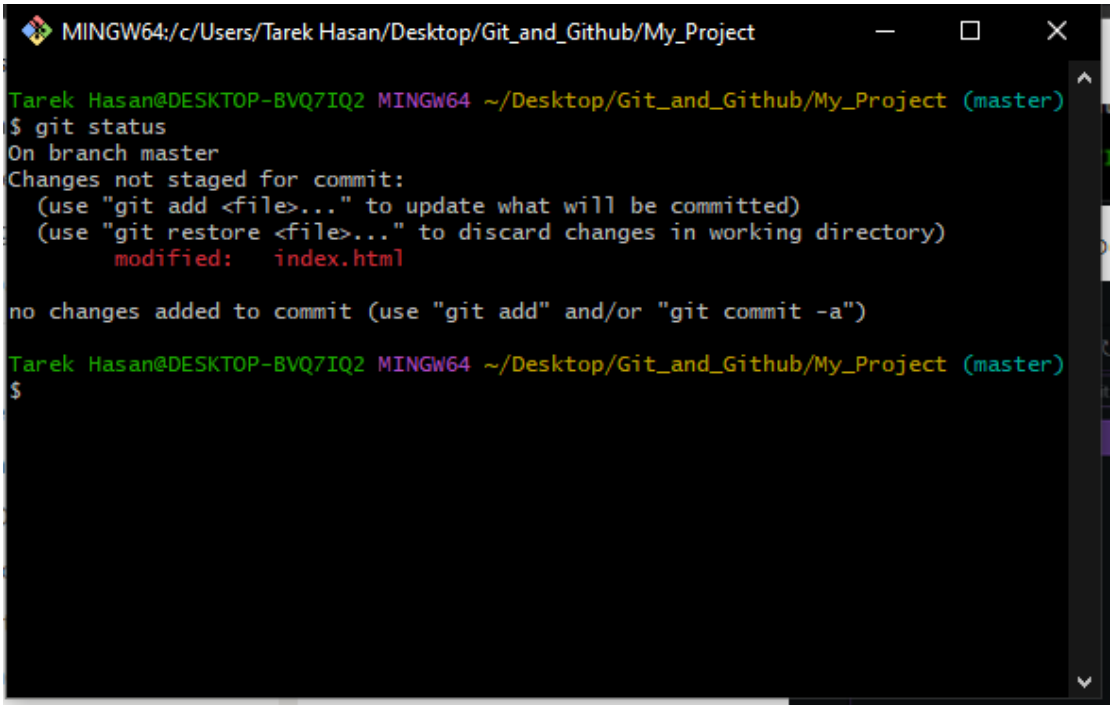
It show “A”, which means index has been added in the staging area. But we haven’t committed yet. Now let’s commit.



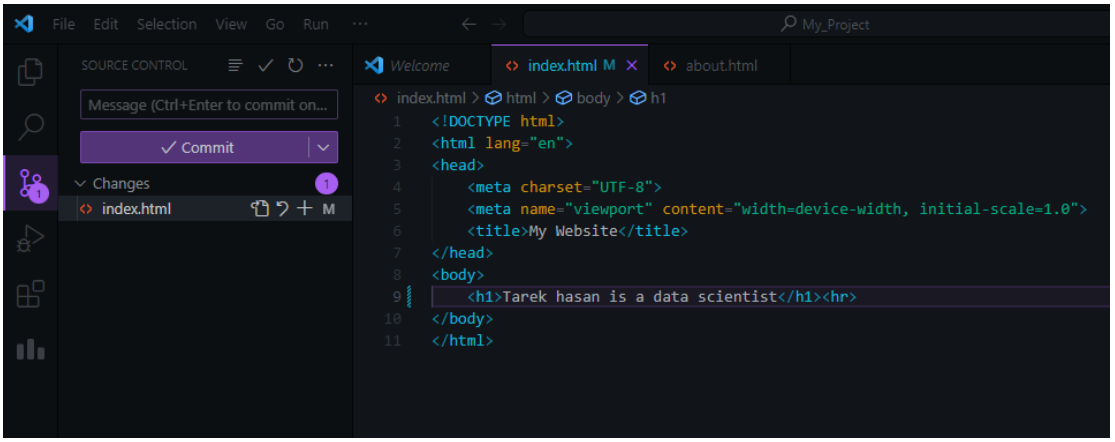
Now let’s see the VS Code status:



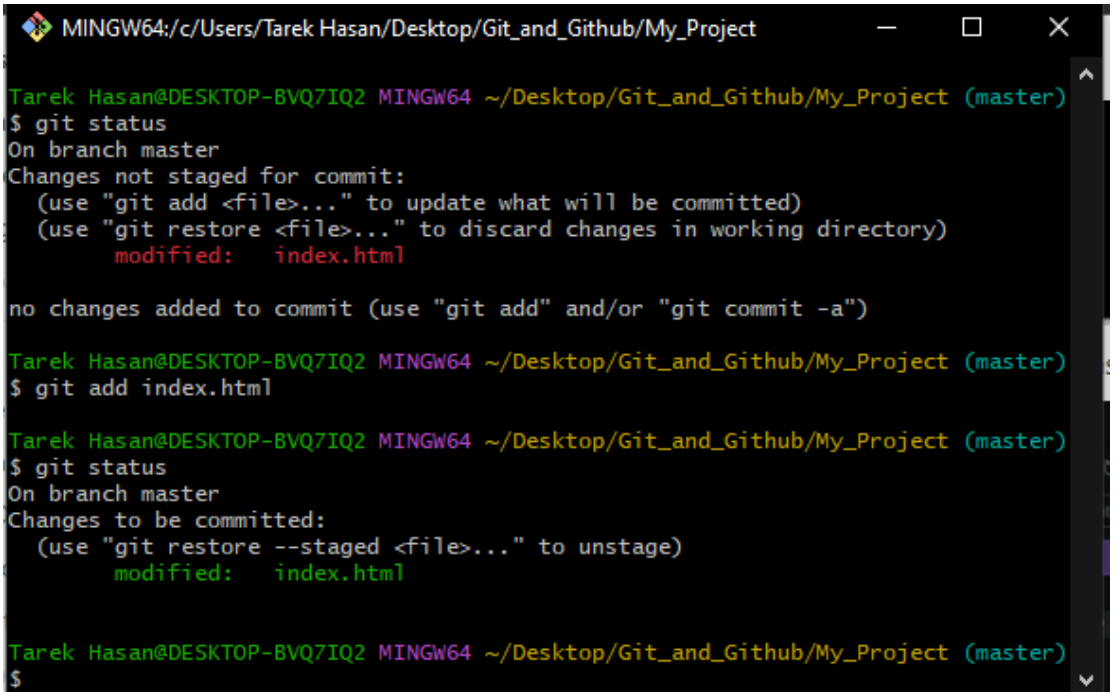
That means we have successfully committed the “about.html” file. Now let’s modify the “index.html” file and see the git status:



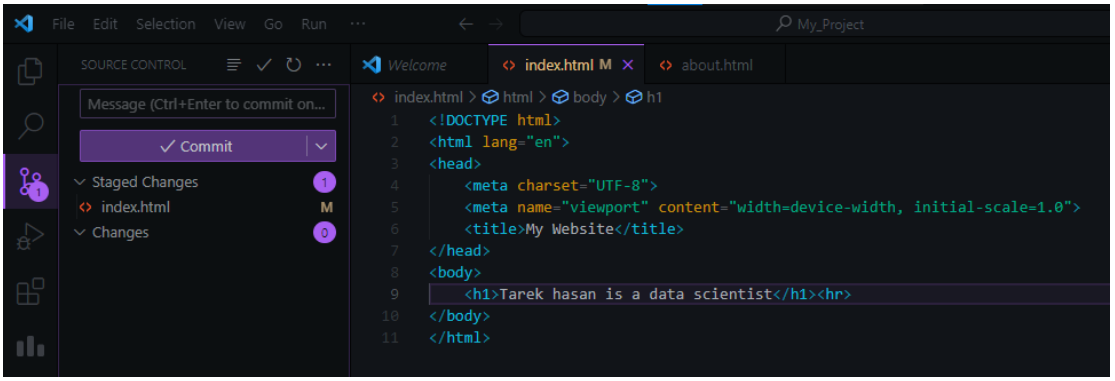
It shows that the file has been modified. Now look at the VS Code status:



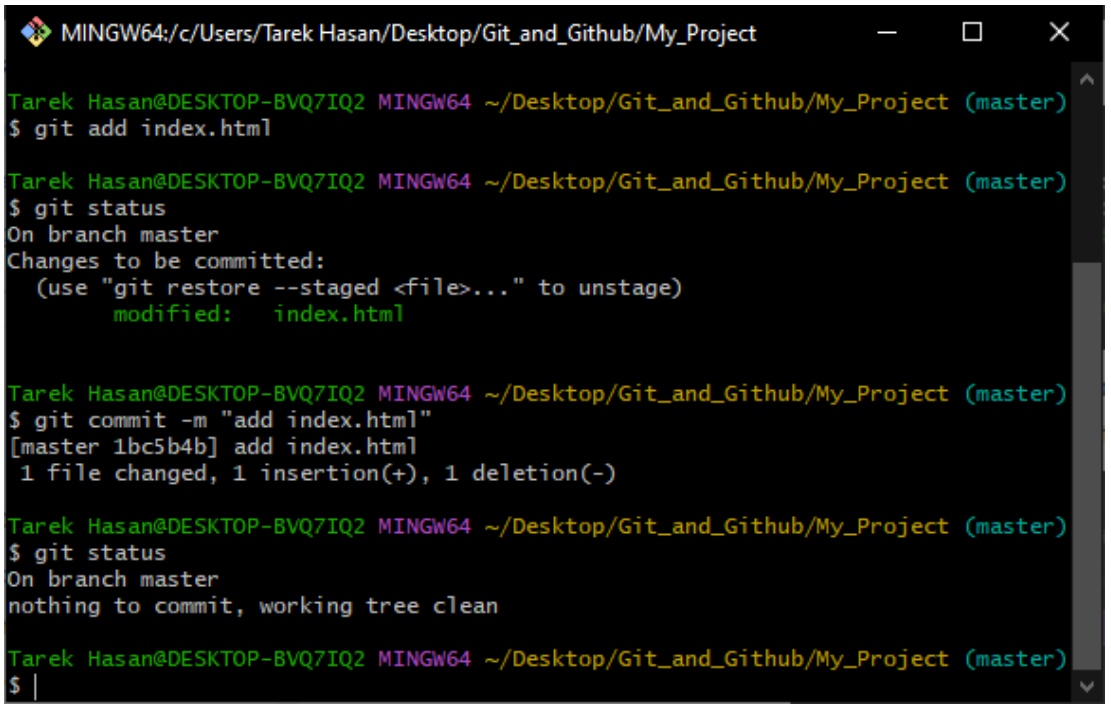
It shows “M” beside the file name which means it’s in the Modified stage. Now what do we need to do? We have to take the file to staging area. After staging the git status is:



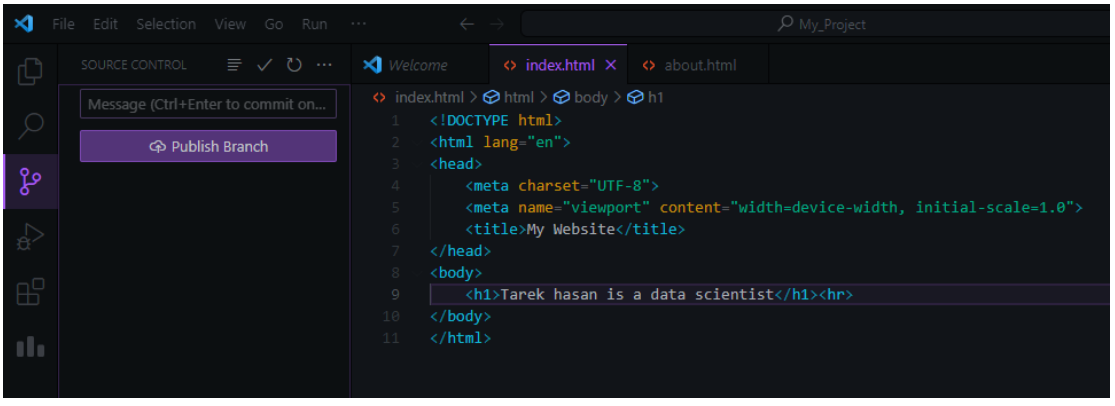
It shows us the status “Changes to be committed” that means the file is in the staging area but we haven’t committed yet. Now look at the VS Code state:



It's still has the "M" sign cause we haven't committed yet. Now let's commit the file and look at the git status:



Now look at the VS Code status:



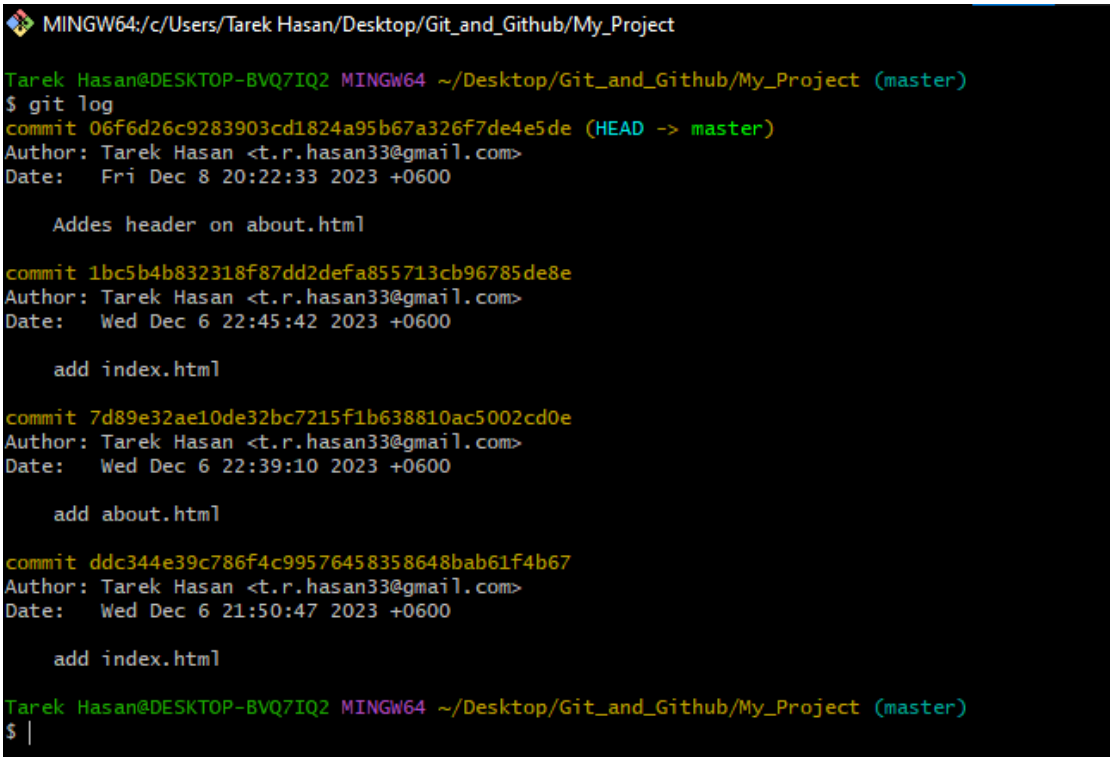
This means we have committed the file. This is the lifecycle of the status of a file.

▼ Logging the previous Commit

▶️ What does logging in git means? It means looking at the previous versions and commits that you made. To look at the commits/version we can use:

> 📌 git log

This command will show the versions and commits.



As you can see, it shows all the changes that I made. The top one is the most recent commit. You may notice there are some text like long secret code.

> ⭐ 06f6d26c9283903cd1824a95b67a326f7de4e5de

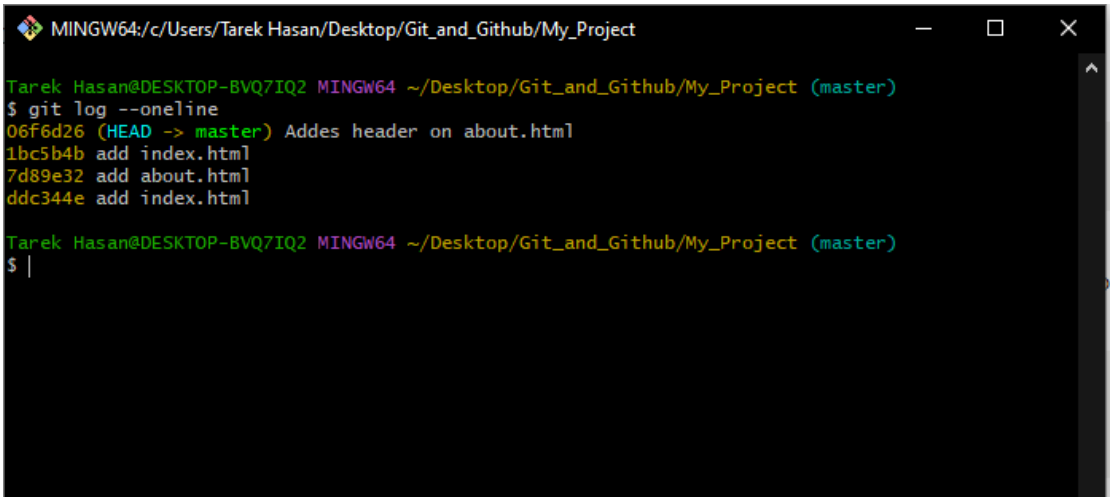
These are hashes of the version/commit. After committing git gives every version of the file a unique hash which is to secure the files as it's almost impossible to decrypt. It also shows the author, his email, date and time.

▶️ **Note:**
To exit the log page you can just press Q.

There is another command which just shows the short hash and the message that you had previously given to the versions while committing.

📌 git log —oneline

It shows output like this:



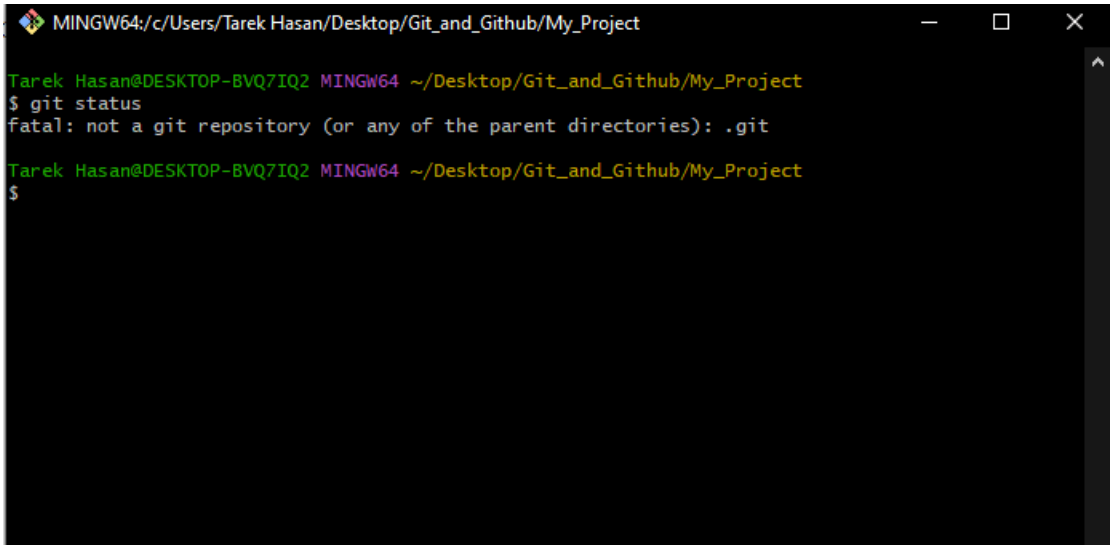
▼ Deleting a Git Repo

▶️ We know that there is a .git named hidden folder in our project folder. It created automatically when we initialized git in our project folder. It stores all the previous version of our committed file. If we delete the .git folder we will loose all the version that also means we cannot commit or add new files before initializing git in the project folder again. To delete a git repository we can use:

📌 rm -rf .git

Here:
rm = remove
-r = recursively [Means delete all the file and folder]
f = force [Forcefully remove even if it's not allowed]

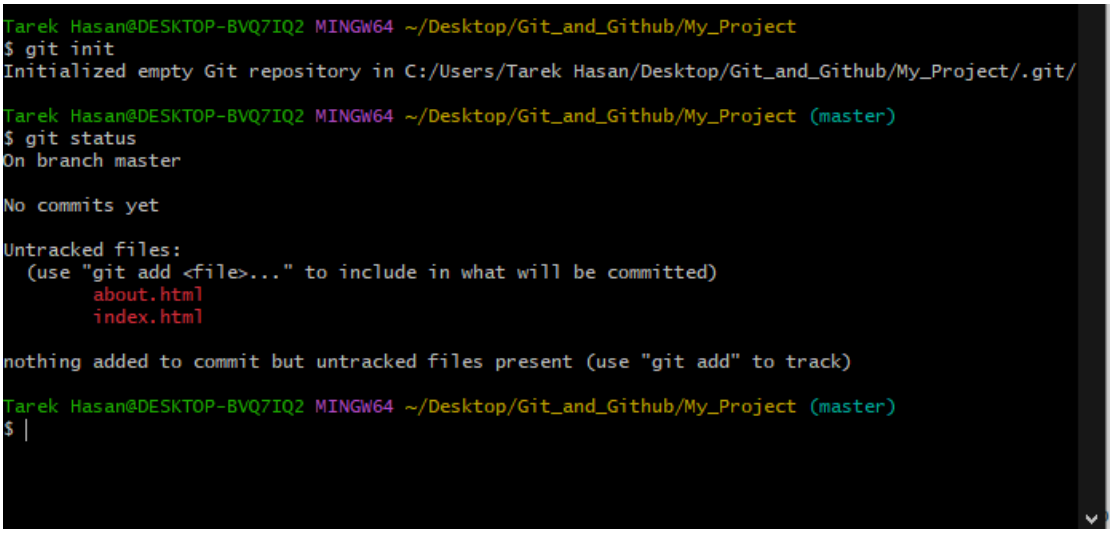
This will remove the .git folder and you will loose the project. After removing and running the git status command you will see something like this:



▼ Additional Git Commands

▼ Skipping the Staging area

▶️ As we know we have deleted the git repo in the previous session. So we first need to initialize the git repository.



After initializing the git repo we ran the git status command and it shows that we haven't staged any file nor we have committed any file. But we want to commit without staging or let's say skipping the staging command. First we will stage those file. We can run

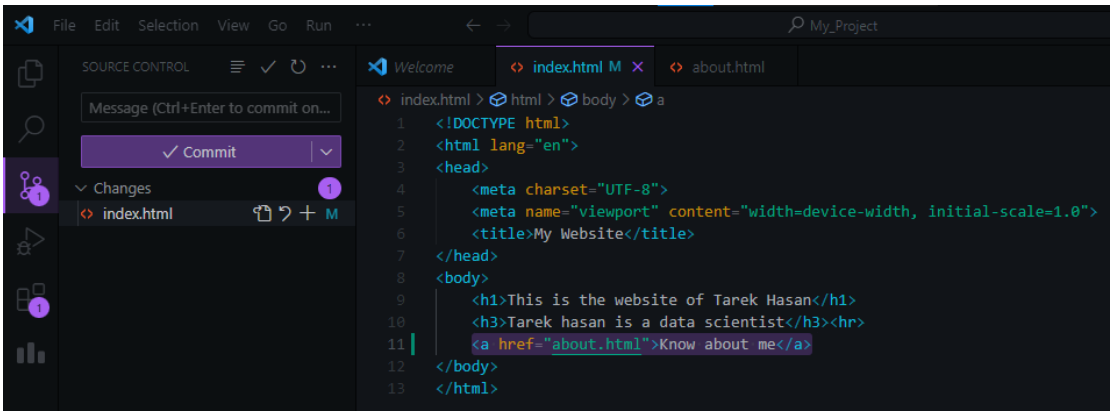
> 📌 git add .

command to add all the files at a time in staging area.

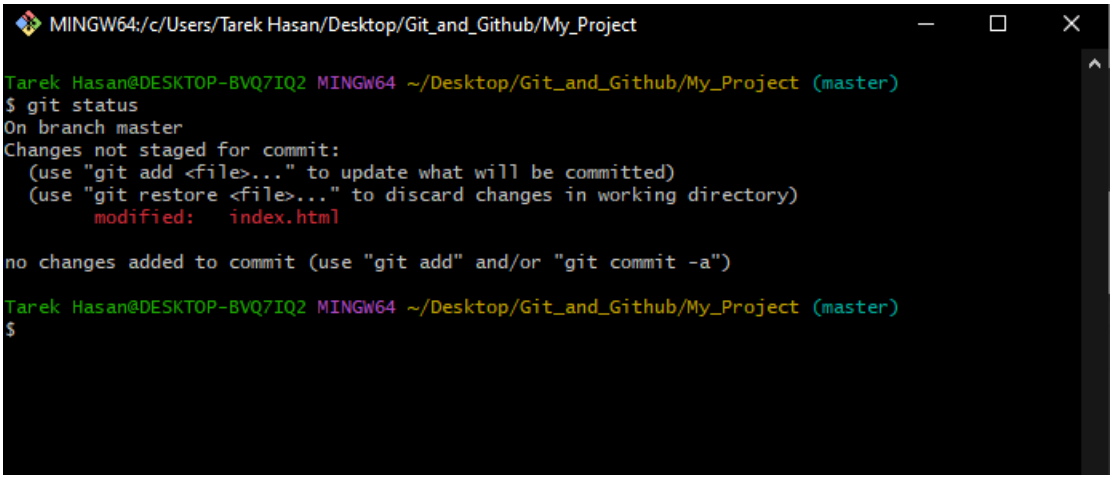
It shows the output like this. Now we will commit those files.

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git commit -m "Initial Commit"
[master (root-commit) 2fe3f0c] Initial Commit
 2 files changed, 25 insertions(+)
 create mode 100644 about.html
 create mode 100644 index.html

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git status
On branch master
nothing to commit, working tree clean

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

Now let's edit one of those files and this time skip the staging part and directly commit the file:



As you can see we have edited the index.html file and added a reference link to the about.html page. Also the side panel shows the M flag which means modified. Let's see the git status too:

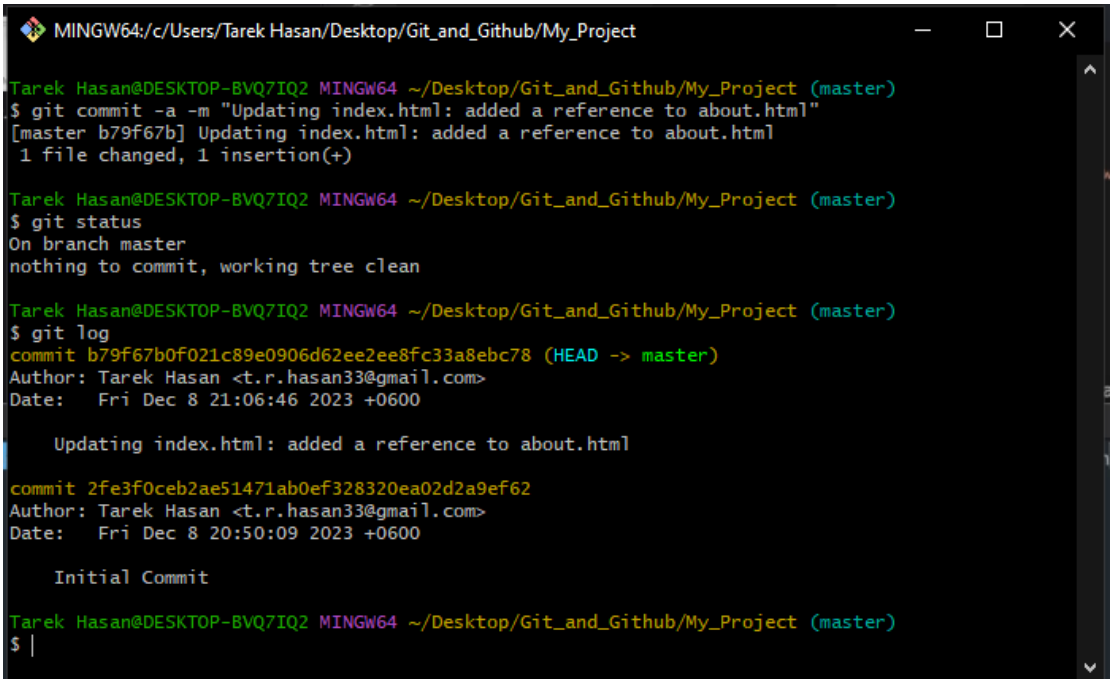


It also shows that the file is modified. Now let's commit the file without going to the staging area. Well we will definitely add it to the staging area but we will do it while committing with one single command. To do that we can run this command:

📌 git commit -a -m <ADD YOU MESSAGE>

Here:
-a = flag -a means adding it to the staging area
-m = flat -m stands for commit.

After running the command:



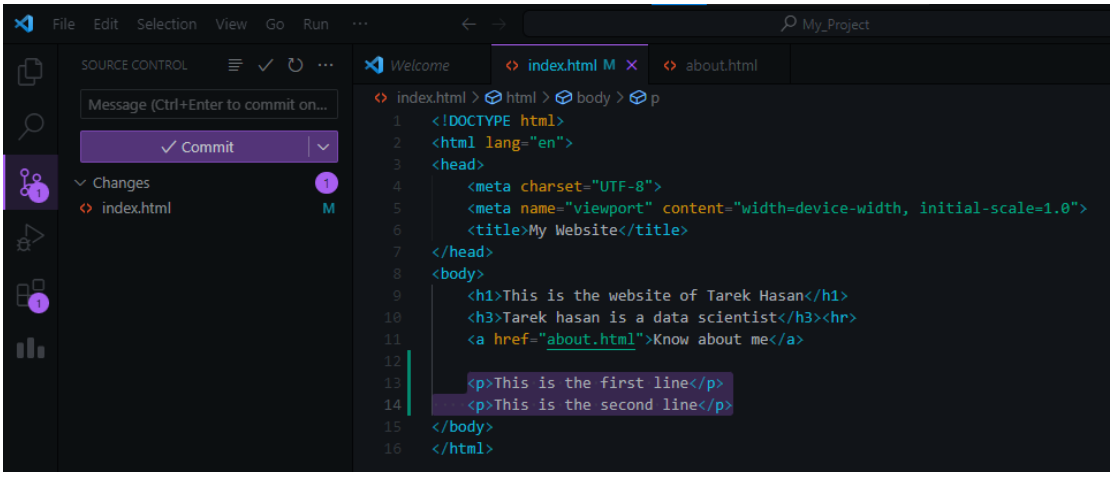
Here it shows what's the git status after doing the commit and also the git log.

▼ **Differences Between Version**

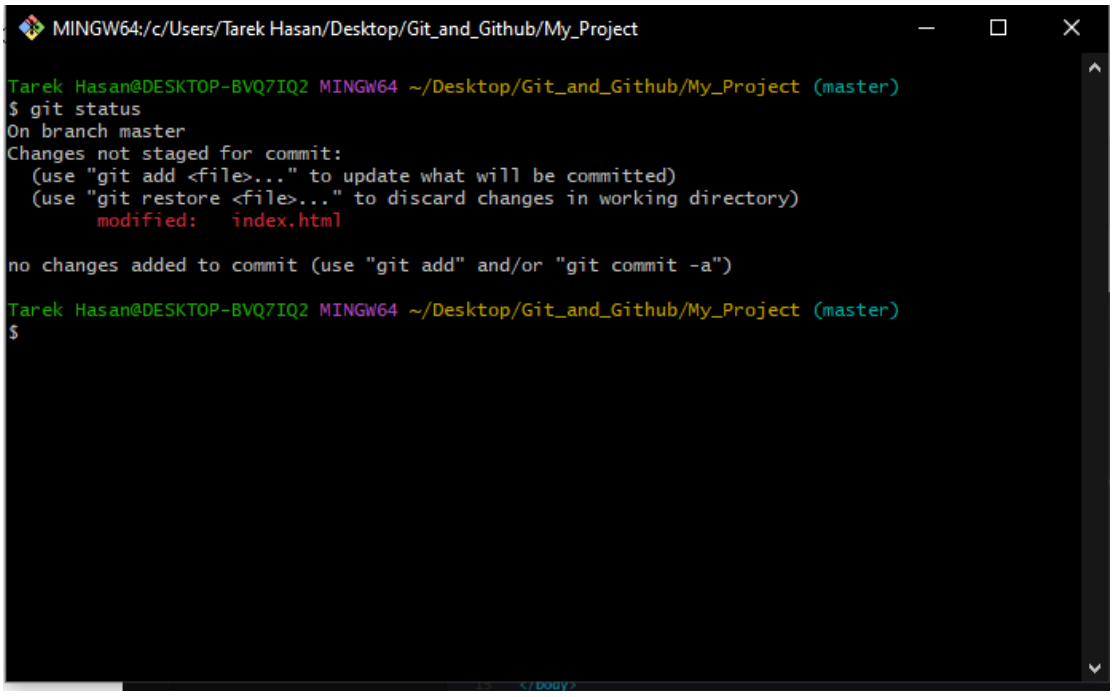
▶️ What does this mean? In our case we have 2 files name index.html and about.html. We have committed them earlier. But after committing suppose we have changed something in the index.html file. Now as we already know this file is untracked and it will be flagged as modified. Here is the difference. This command will show the difference between the committed version and the modified version means we can see what did we modify from the previous version. Here is the command:

📌 git diff

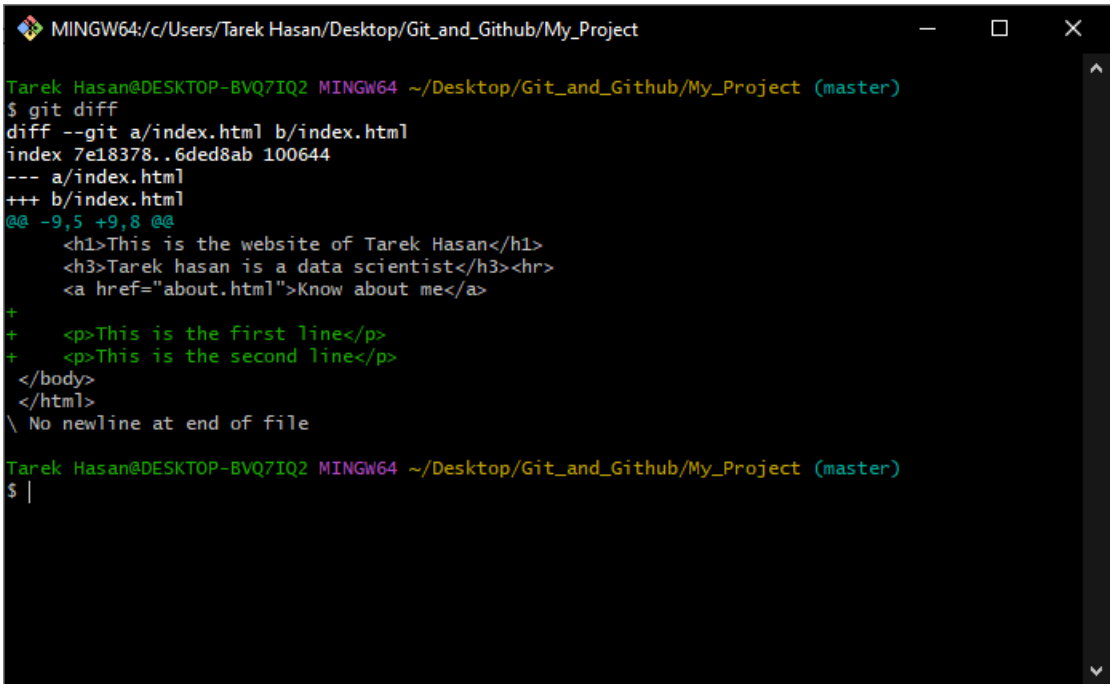
Now we are going to modify the index.html file.



We have added to paragraph line in the bottom code. As you can see this file is flagged as modified.

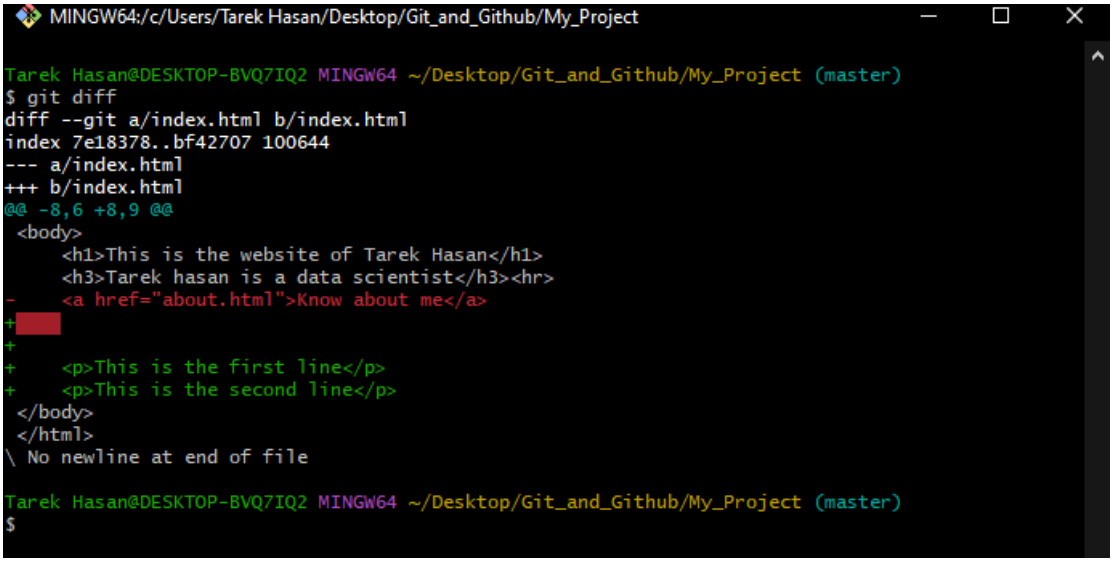


The git status command also says that the file is modified and not staged for commit. That means the file is untracked. Now lets run the command and see what happens:

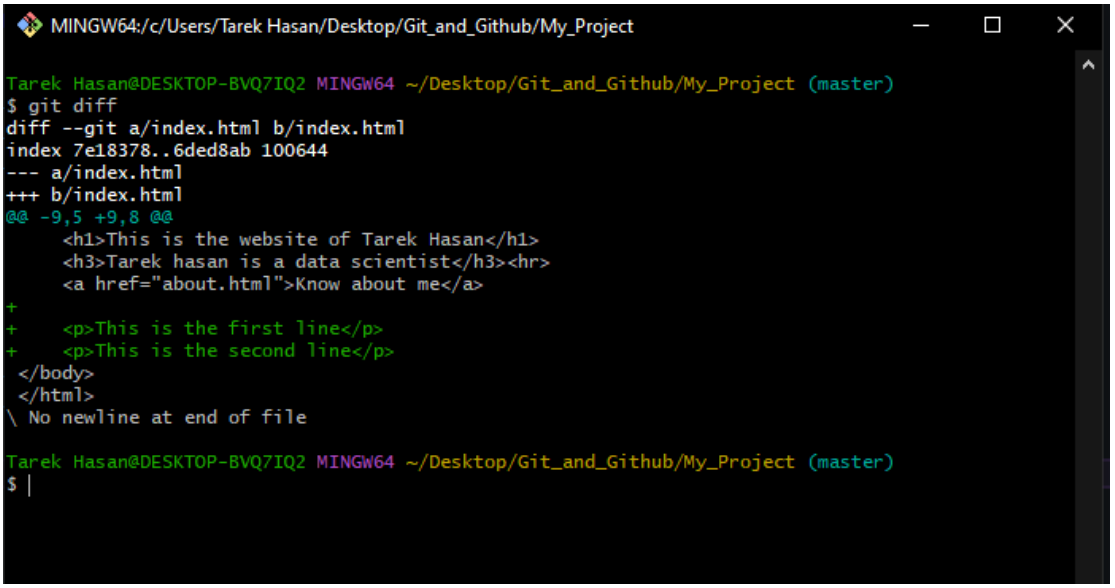


Here we got 3 “+” signed lines with green color writing. That means these are the new lines that we have added. What happens when we remove a line from the current untracked file?

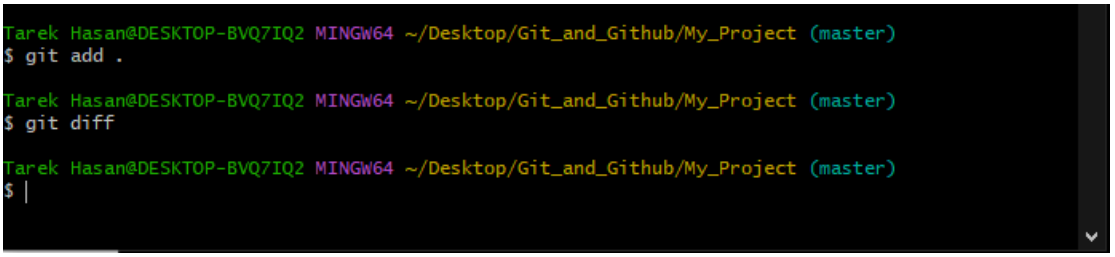
We now are going to remove the reference line temporarily to show the example and look at what git diff shows us:



Look at the red line. As we have removed the reference link line it shows it as red as we have the line in the committed version. After adding the line again it will show nothing in red like this:



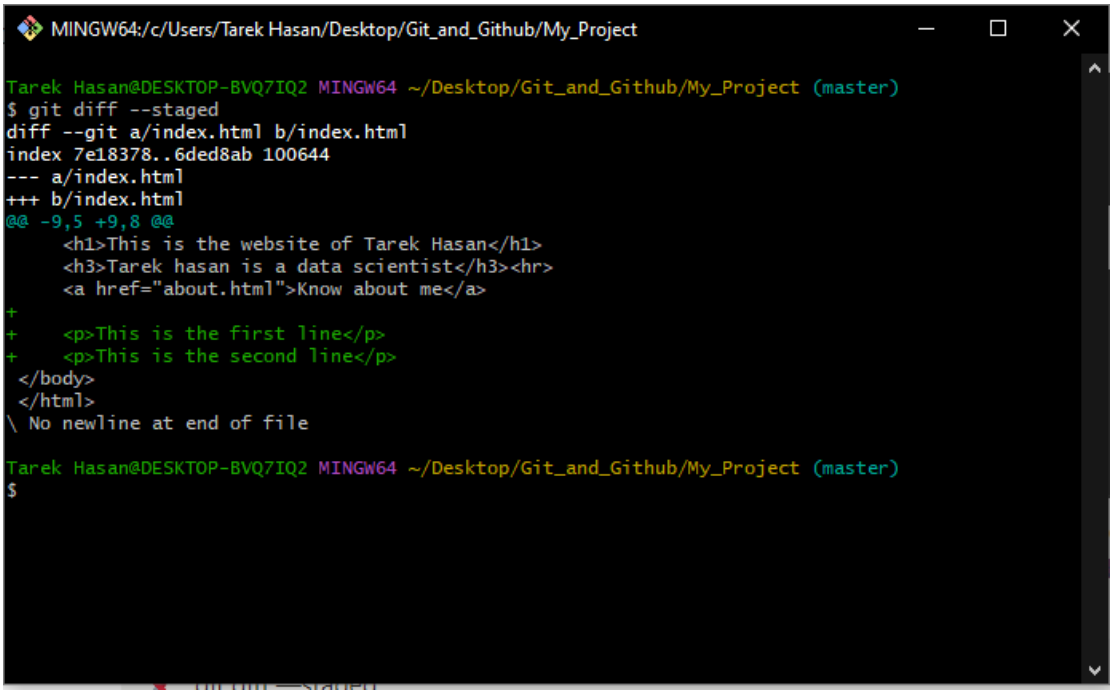
Now let's stage the files and see what git diff shows us:



It shows nothing. Cause the file is not untracked anymore. It is in the staged area. But we can also see difference between the committed version and the version that is in staged area with using a flag in the command. The whole command is:

> 📌 git diff —staged

Let's see what it shows us:



It shows the differenced between staged file and the file that is already in the committed version.

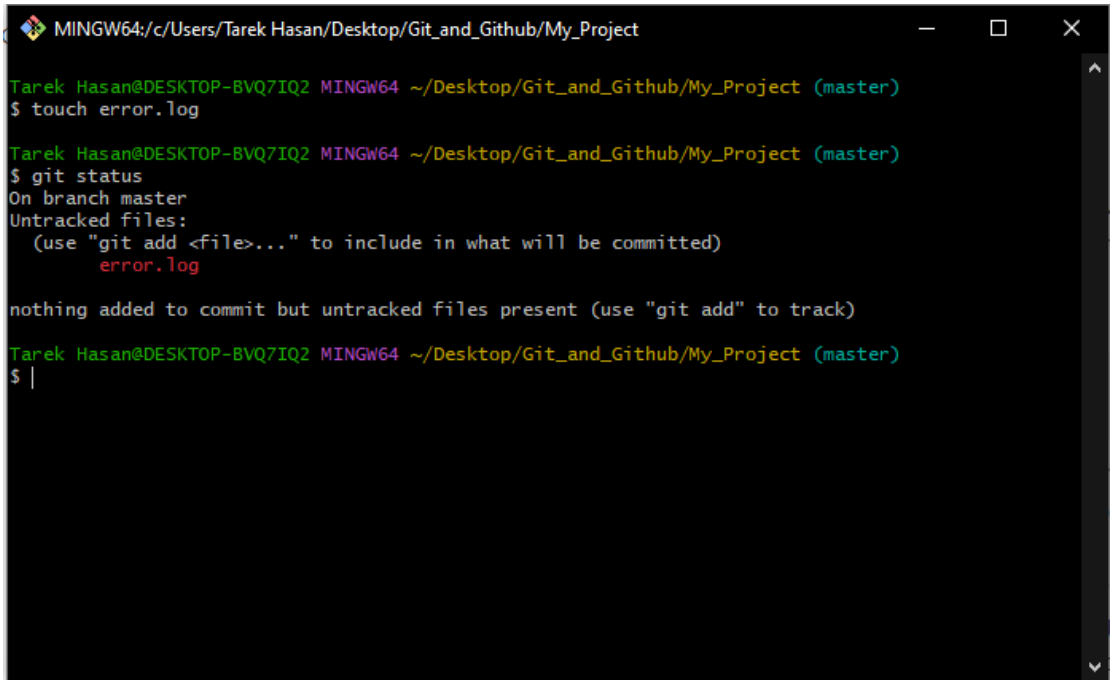
▼ Ignoring Files

▶️ Suppose, in your project folder there are few files like index.html and about.html as well as some other unnecessary files. The problem is when you stage all file at once or commit changes it will commit all the files. But you don't want that. You want to ignore those unnecessary files. There is a way to do that.

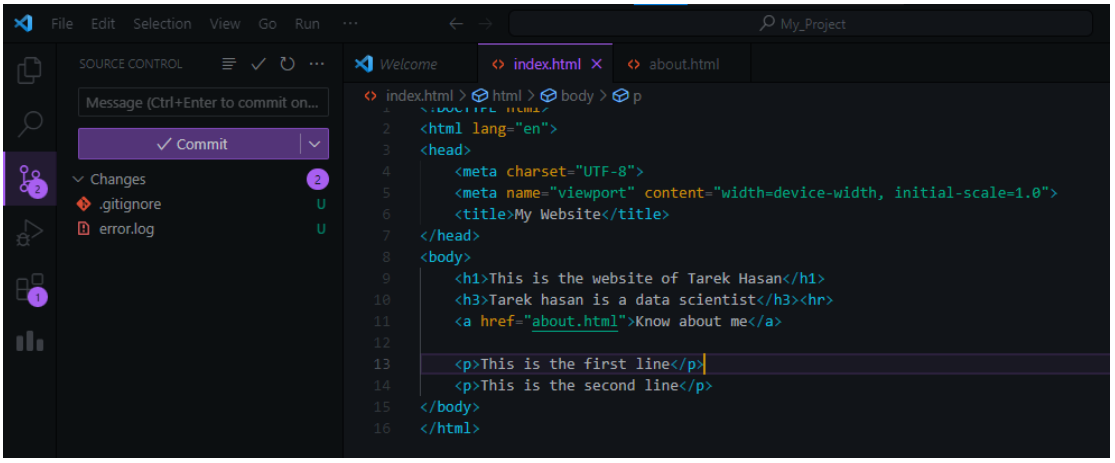
> 📌 .gitignore [It's a file]

You are already familiar with a hidden folder named .git. This is also a hidden file named .gitignore where you can write all your files or a kind of pattern and those will be ignored. Now let's look at it's functionality and how it works with practical implementation.

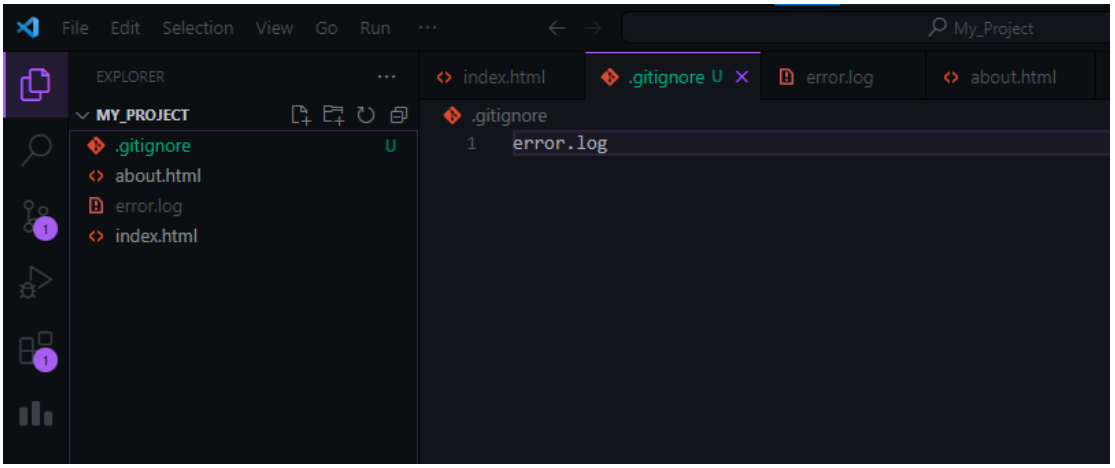
We have 2 files in our project directory named index.html and about.html. We are going to create another unnecessary file named error.log. We are going to create the file using touch command and look at the git status.



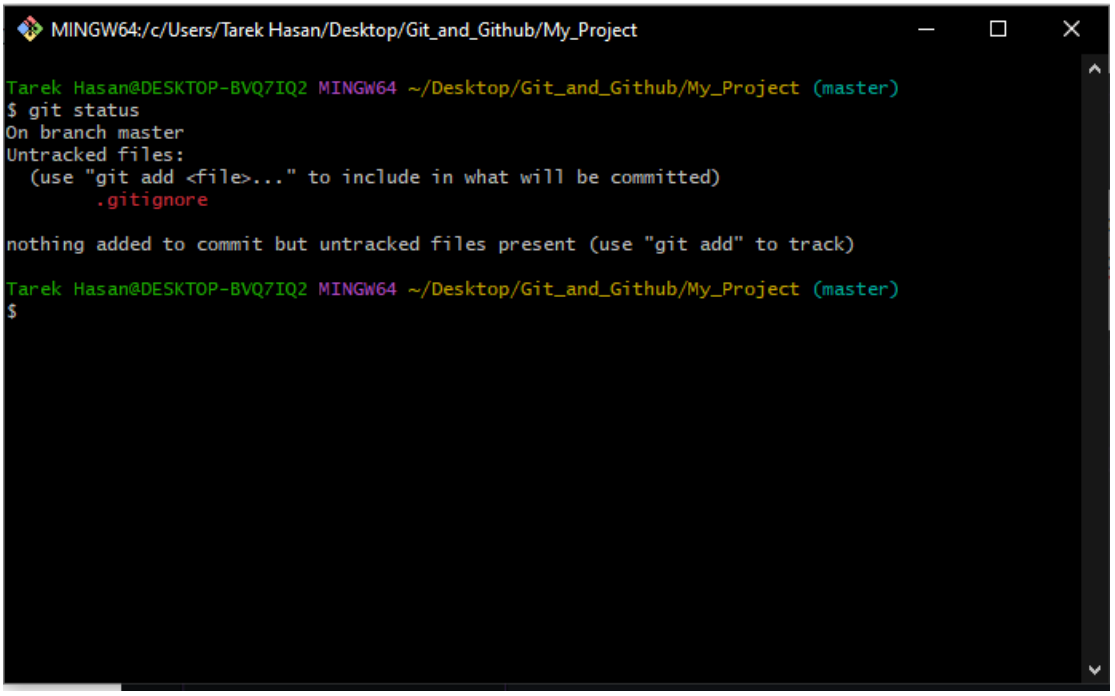
It shows that we have error.log file and it is untracked. But we don't want it to be tracked. We don't want it to be in staged area or to be committed. Now we are going to create another file named .gitignore and open it in VS Code.



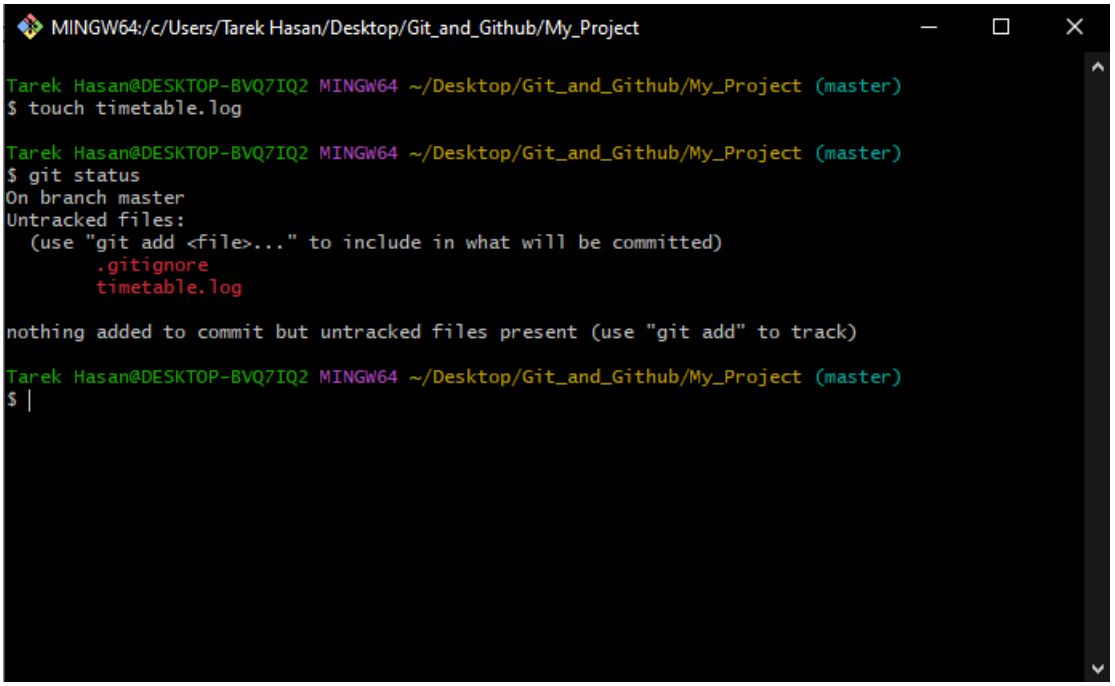
Look at the side panel. error.log and .gitignore files are flagged untracked. Let's talk about this later. Now we are going to put the name of the file we want to ignore in the .gitignore file which is in our case error.log.



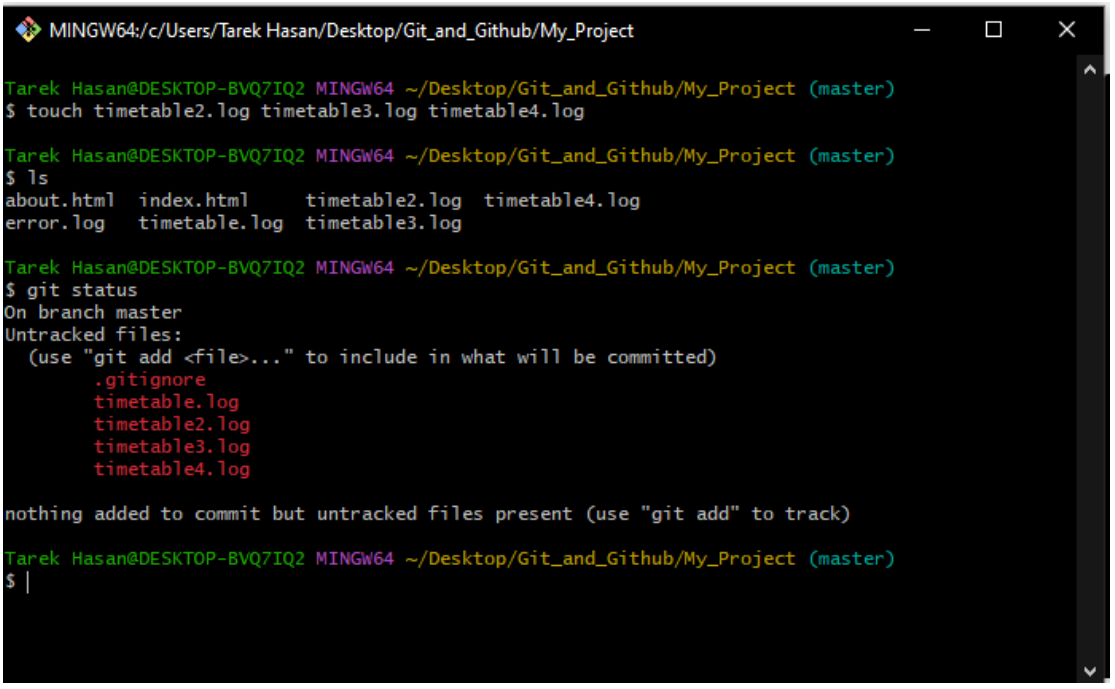
Now let's look at the git status:



It says there is a untracked file named .gitignore. Why is showing as untracked? Cause it's a file and we need to commit this file. Then git will ignore the files which is written inside the .gitignore file. But where is error.log? Why isn't it showing? Cause we have written the file name inside the .gitignore file. But if it's already working then why do we need to commit the .gitignore file? Cause git needs to keep track of this file to keep ignoring new files that we are going to add here. Let's create another file and check the git status if we don't add it to the ignore list:



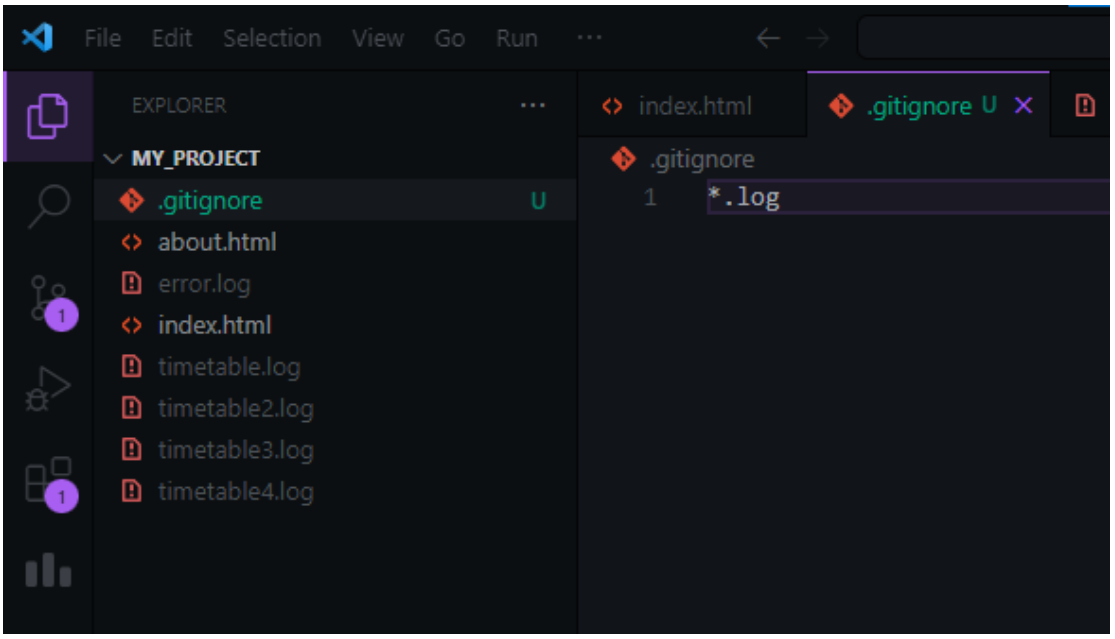
Now it shows both of the file as untracked. But it isn't showing error.log cause we added it to the .gitignore list. Now let's create 3 new log file named timetable2.log, timetable3.log, timetable4.log.



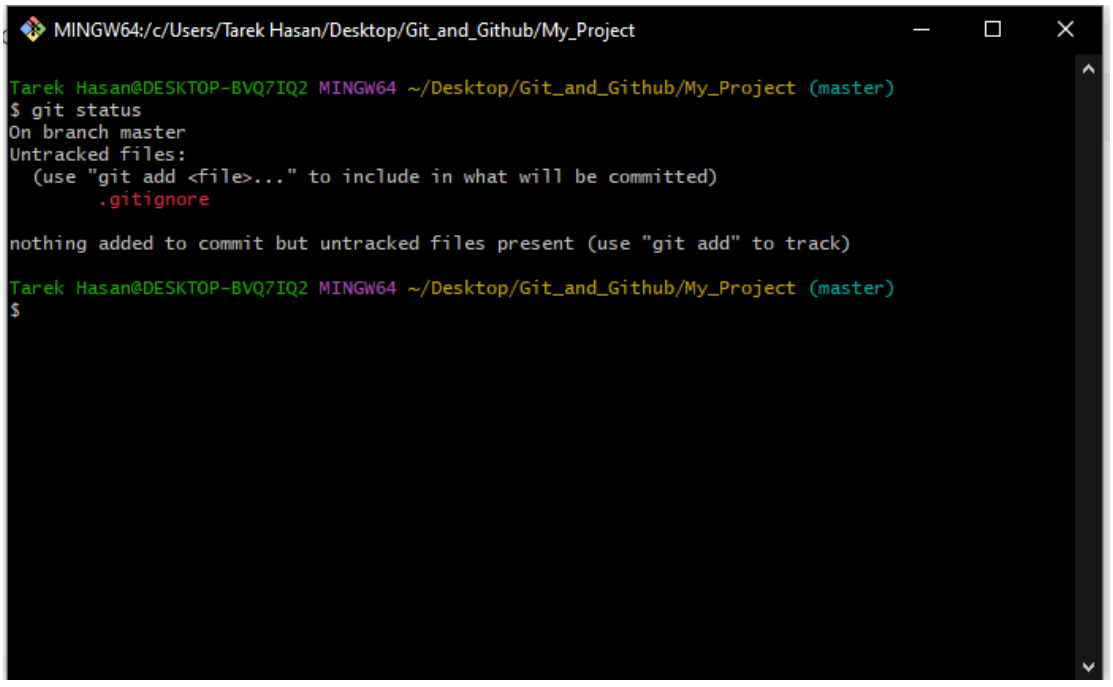
Look at the git status. It is the file error.log that is in the ignore list but not ignoring these additional files. Now is adding them one by one in the ignore list the efficient way? Obviously not. What can we do is we can use regex, means we can ignore them by their file extension. So we want to ignore all the files with .log extension. We can use below method to write extension to ignore in the .gitignore file

📌 *.<EXTENSION_NAME>

Look at the VS Code example:

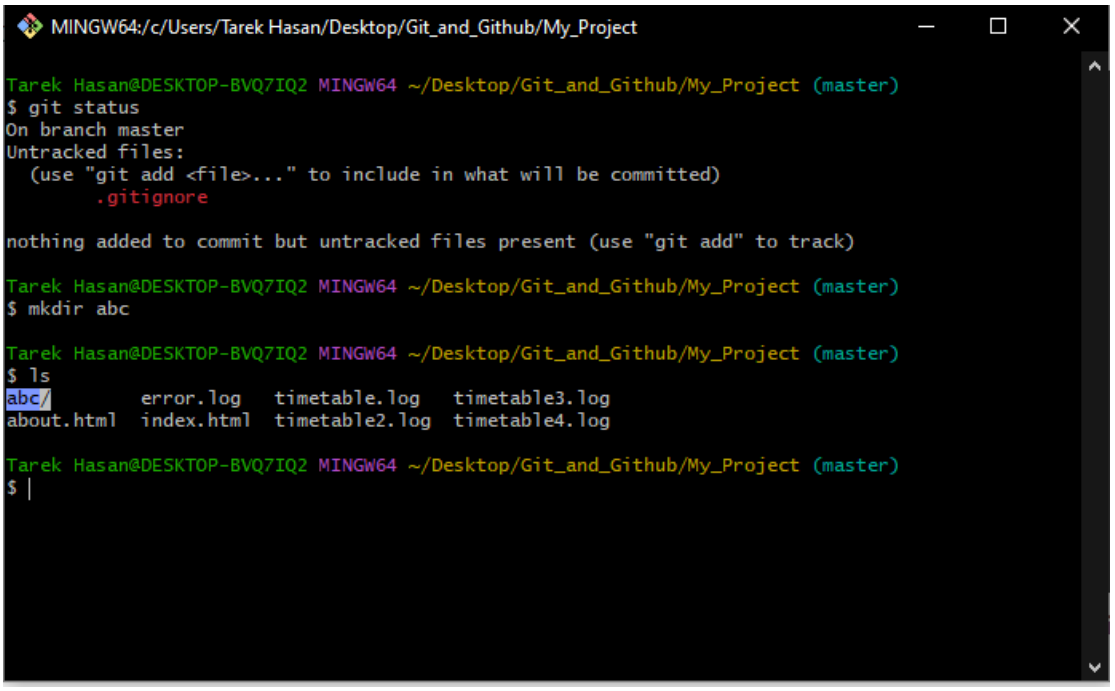


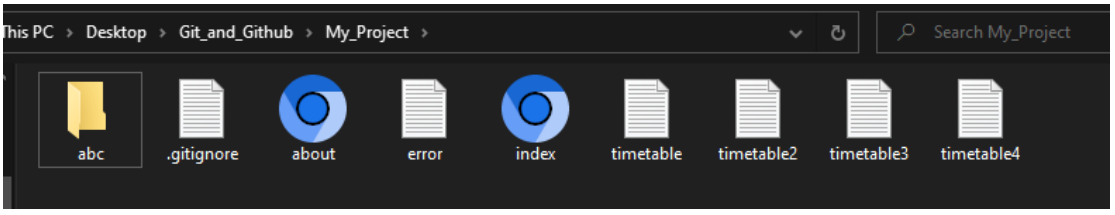
Now let's look at the git status:



It isn't showing all those files as untracked now. From now on every file with .log extension will be ignored. Now suppose you have a folder inside you project folder

that you want to ignore. How can we do that? At first let's create a folder inside the project folder using mkdir command. The folder will be named abc.

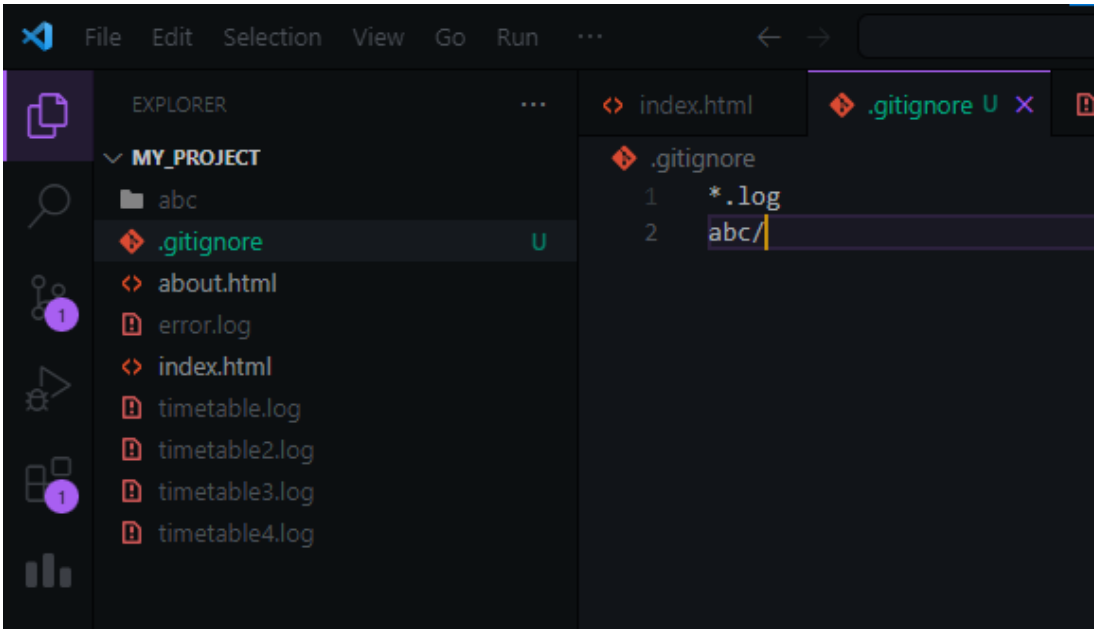




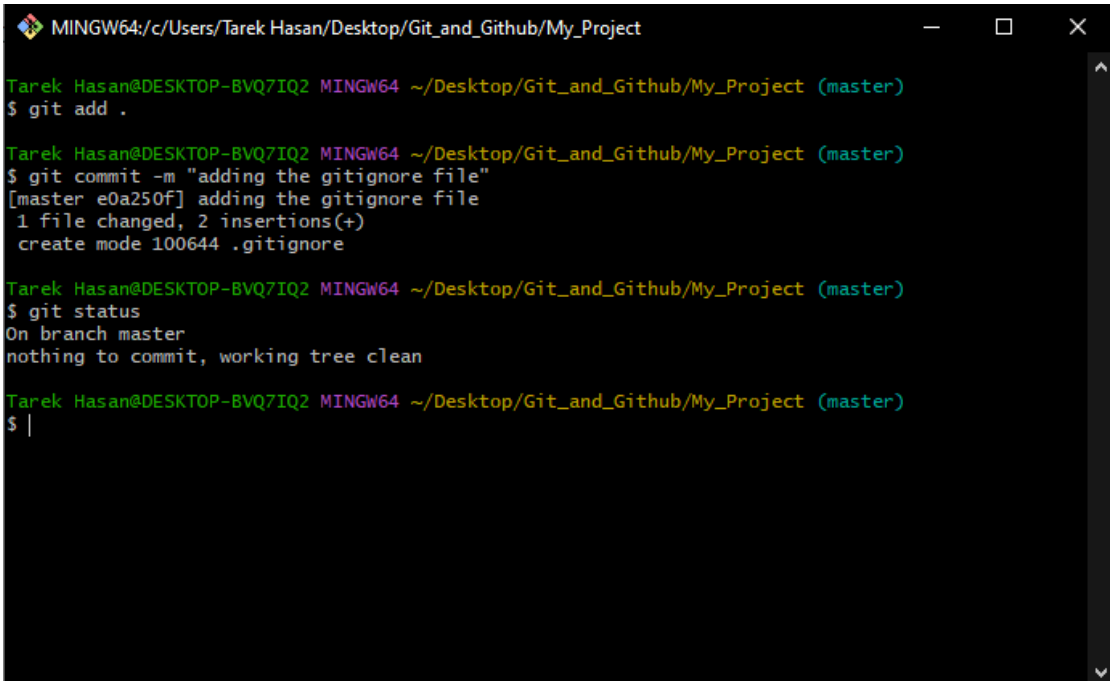
As you can see we have a folder named abc now. How can we ignore it? We just need to put the folder name in the .gitignore file with the below method:

📌 <FOLDER_NAME>/

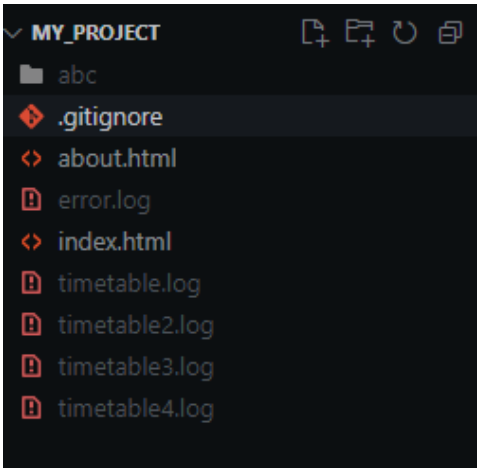
Look at the VS Code example:



Now let's stage and commit the files and look at the git status:



So all of them are getting ignored. Look at the VS Code side panel. All the files that are being ignored are grayed out:



We are now going to add a random.txt file using touch random.txt command for our next session. Also remove all the additional timetable.log files keeping the real one.

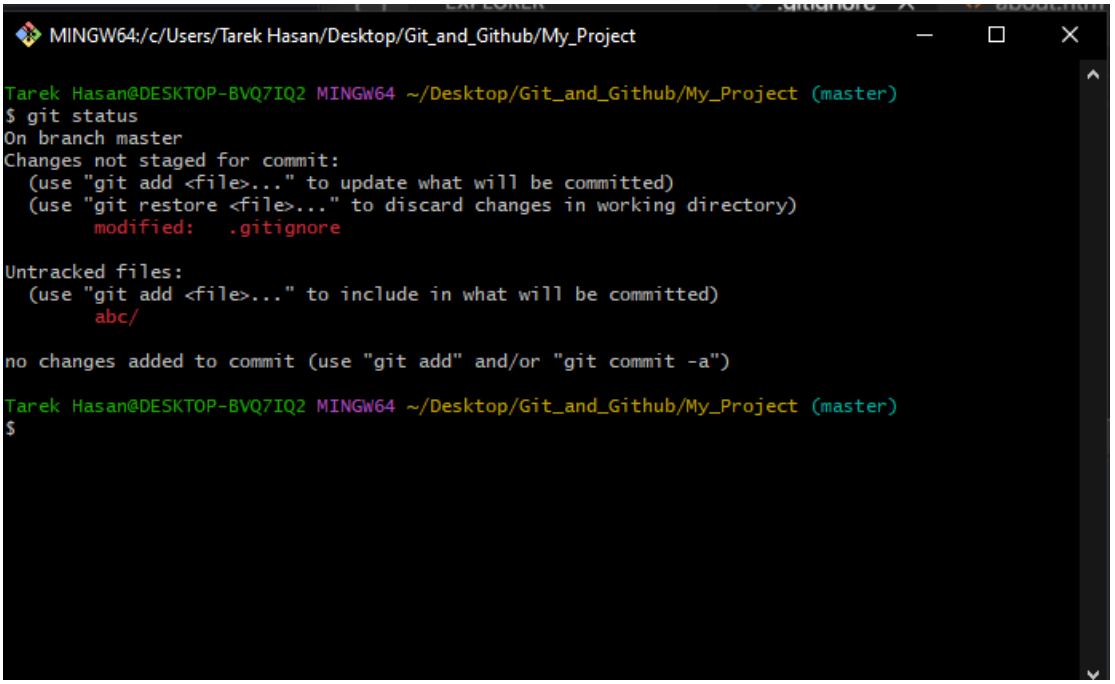
▼ Moving and Removing Files

▶️ We already know the commands to move, remove and copy/paste files. We can also implement the commands here. But there is a catch. When ever we use those commands we will again need to stage the files explicitly. But to get rid of the hassle we will use the commands with git. Using commands with git will automatically stage the files when we remove/move them.

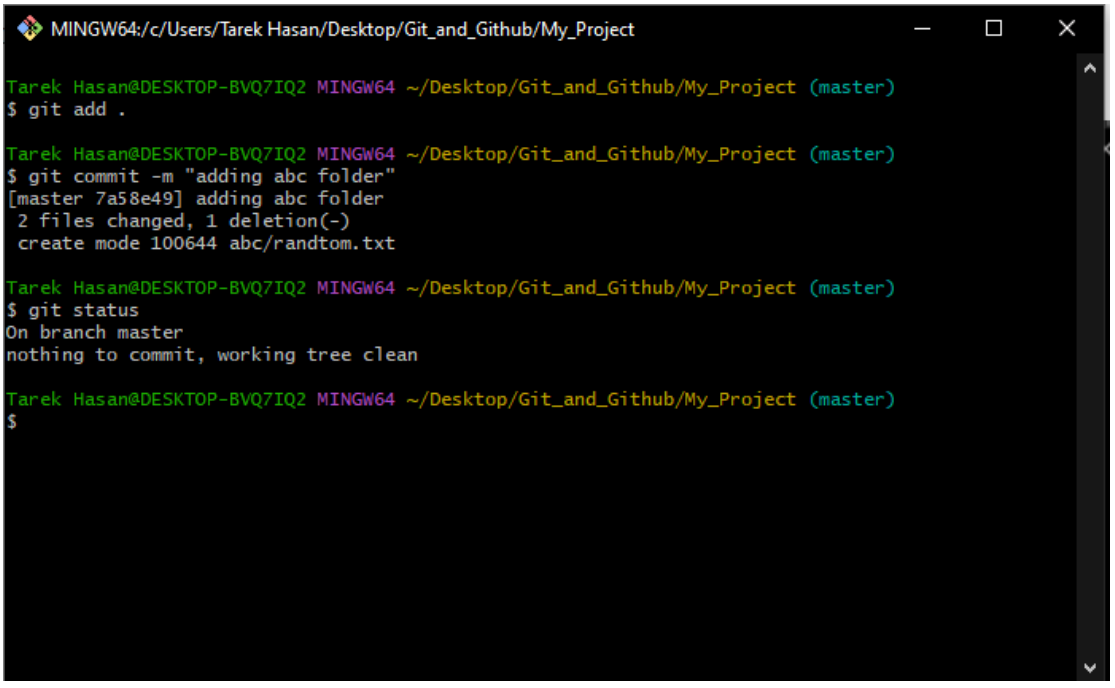
📌 git rm <FILENAME>.extension [To remove a file]

📌 git mv <FILENAME>.extension <NEW_FILENAME>.extension [To move files]

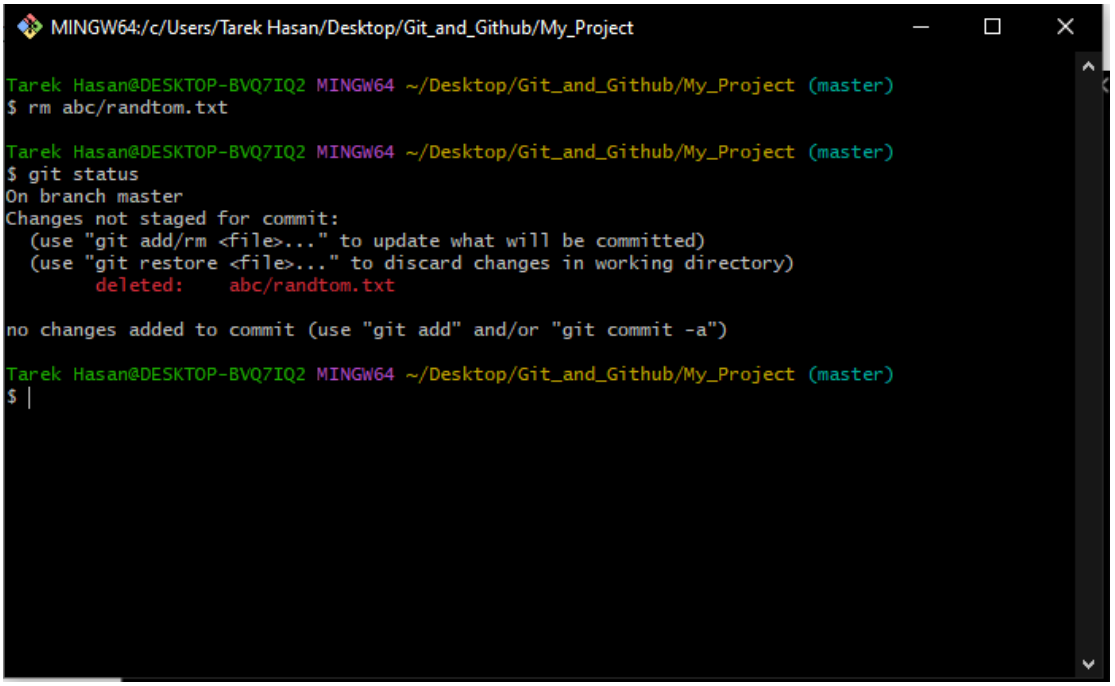
Now let's begin with the practical example. Firstly we will remove the abc folder from ignore list.



As you can see it's showing as untracked that means we are not ignoring it. Now stage and commit all the things and look the git status:

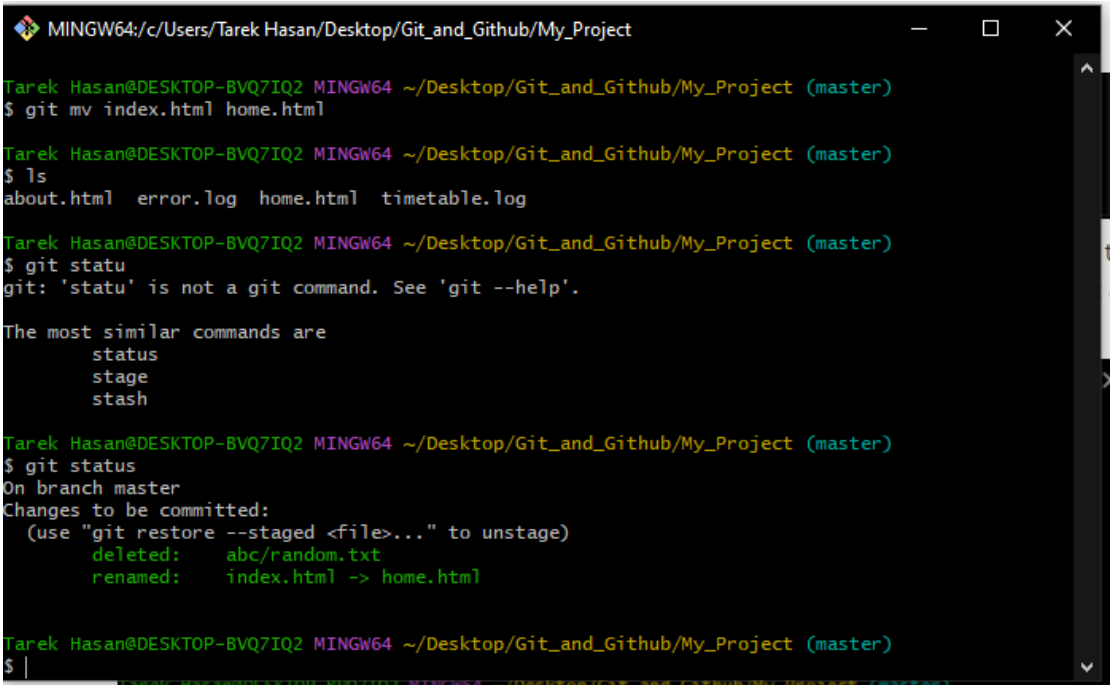


So we have added everything. We already know we have a file named random.txt inside the abc folder. Now let's remove it using the regular command without using git and look at the git status:

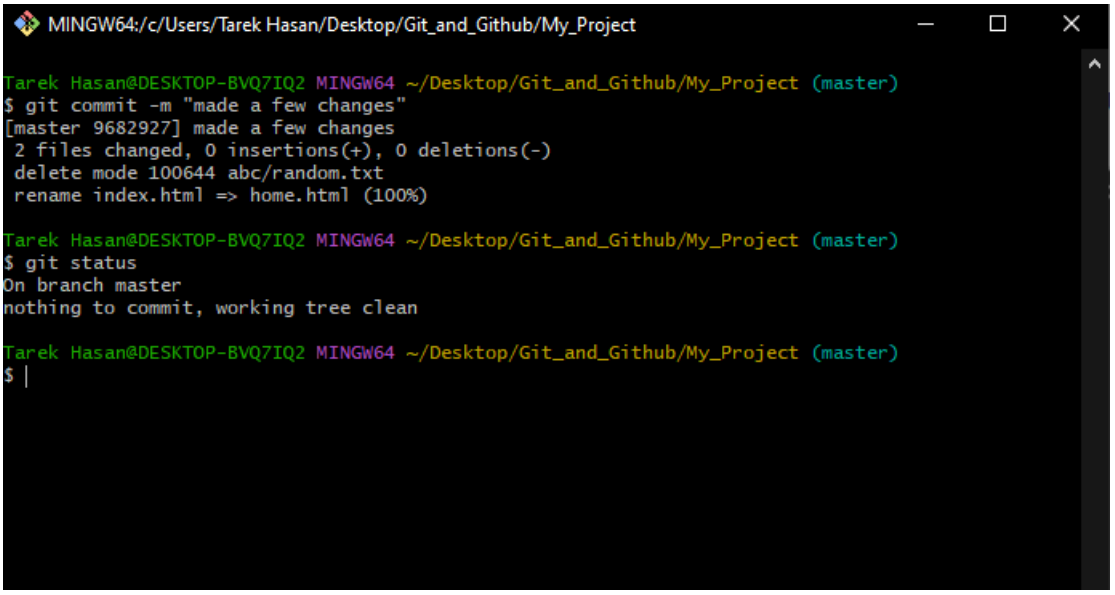


It is saying that changes not staged for commit. And the text is in red color. So we have to stage is separately. Now let's re add the file in abc folder and run the command with git and look at the status:

As you can see now the line is in green and it only says that changes to be committed. That means it's already staged and we don't need to stage it explicitly. Now let's move the index.html in the same project directory using a new name home.html and look at the git status:



You can see we have renamed the file to home.html and it's already staged because of the git. Now we don't need to stage them. We can just commit. Let's commit and see the status:



That's it.

▼ Untrack a Already Tracked File

▶️ We already know how to ignore files. But there is a catch. When you commit it starts to track all the files. Means it already has a version of those files. If you add any files from the files you have committed git will still keep tracking the file.

Cause git has a version already of that file. To solve this we need to delete the cache of the file to start ignoring it. We can do that by running:

> 📌 git rm —cached <FILENAME>

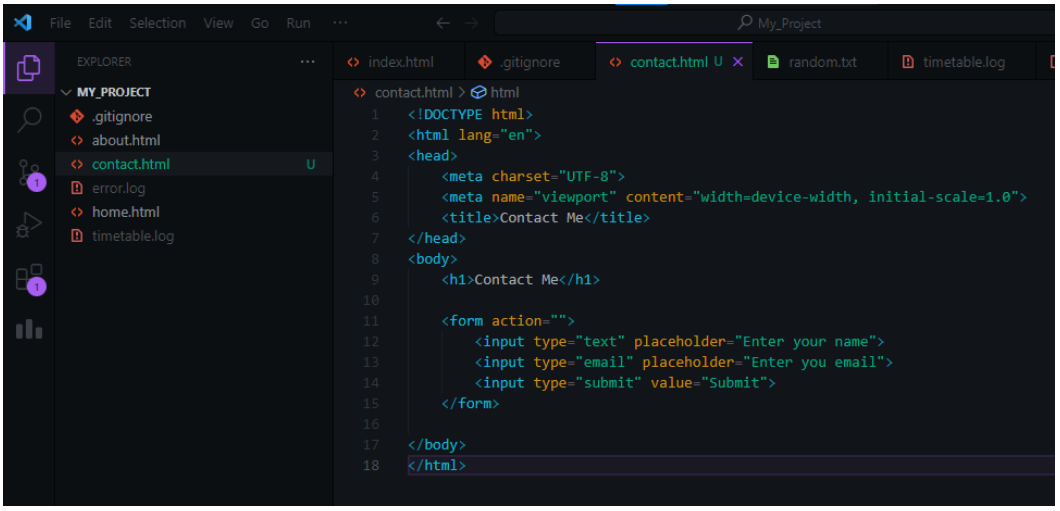
▼ Undoing Somethings

▼ Unstaging Files

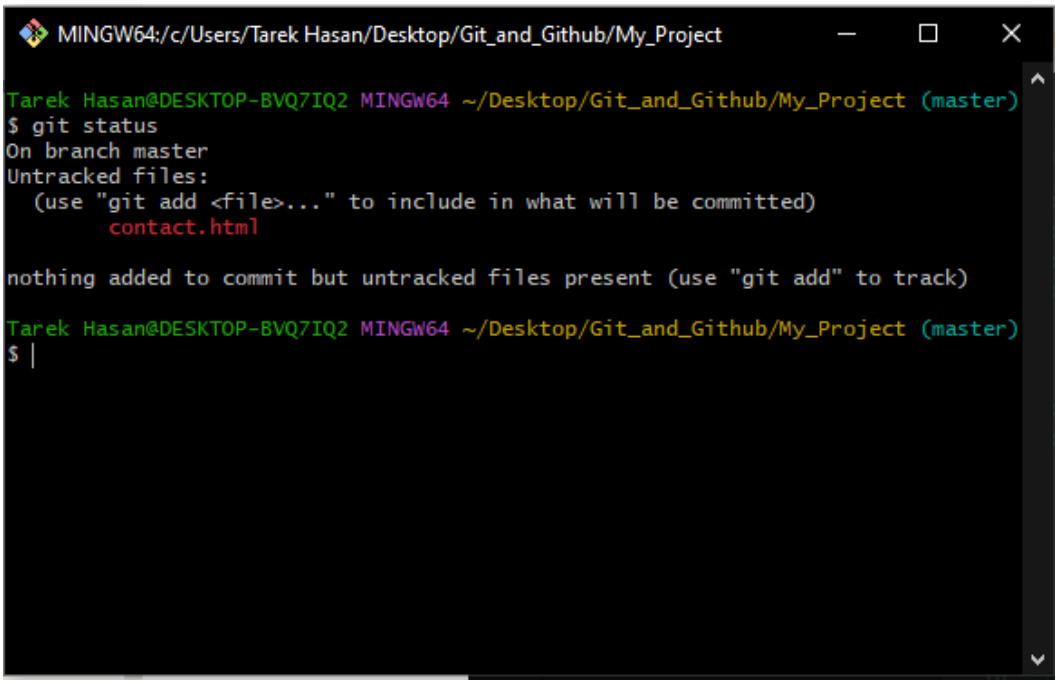
▶️ Suppose you have written a code which you thought earlier that it works file so you have already staged the file. Means you have added the file in the staging area. Now how to unstage the file? Here is a command for that:

> 📌 git restore —staged <FILENAME>.extension

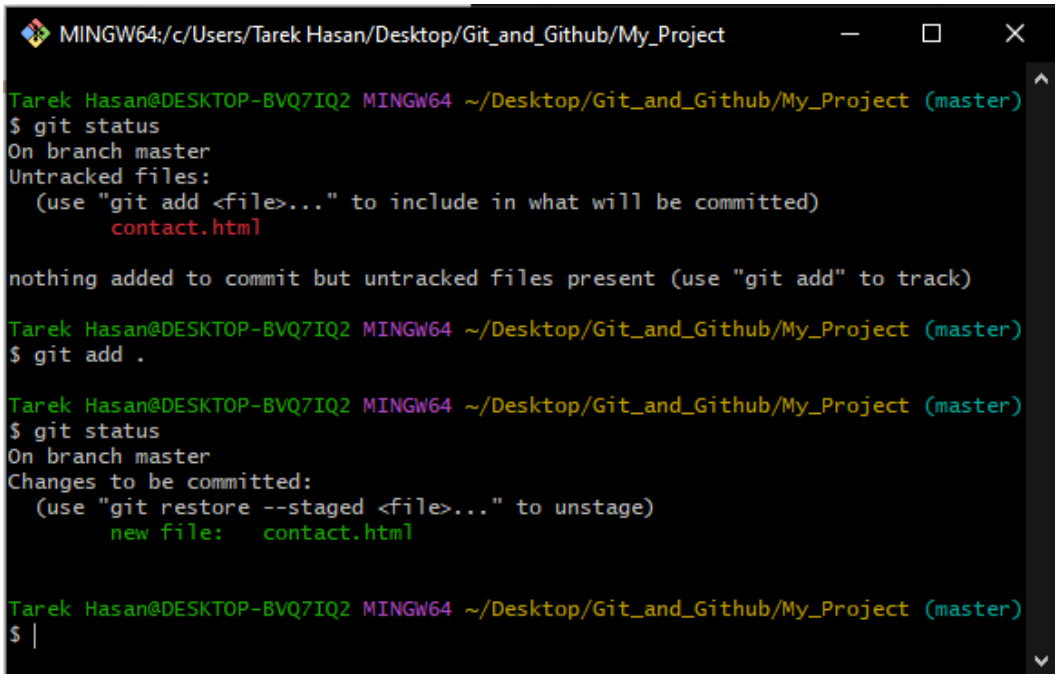
To look at the practical example let’s create a new file named contact.html and let’s look at the VS Code window:



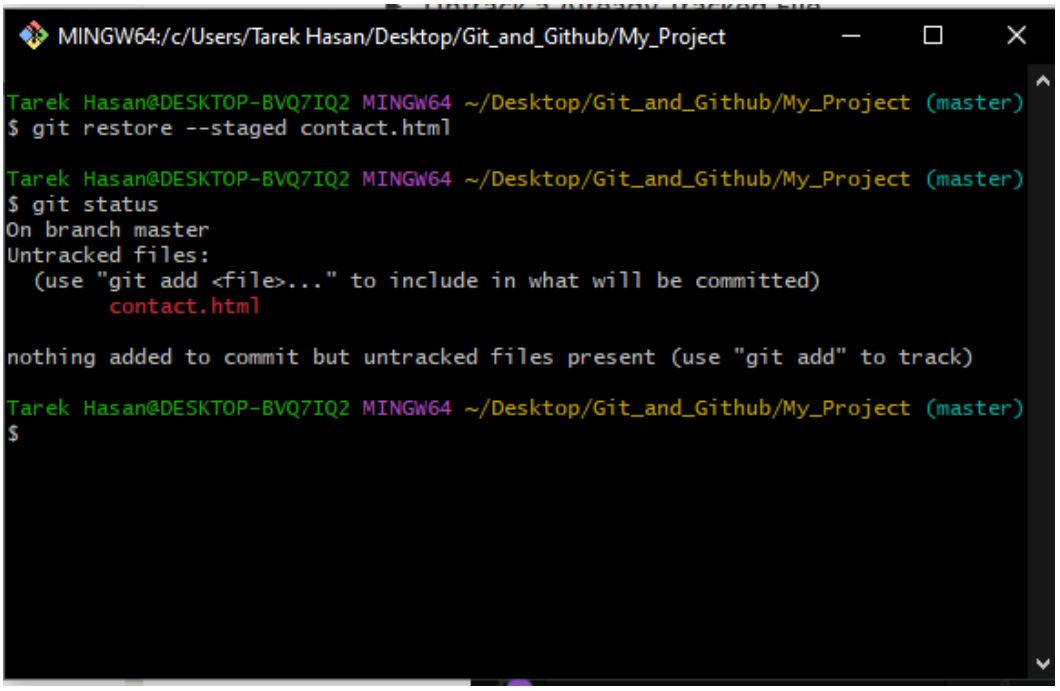
Now let’s look at the git status:



Now let's stage the file and look at the status:



Now let's unstage the file with the command and look at the status:



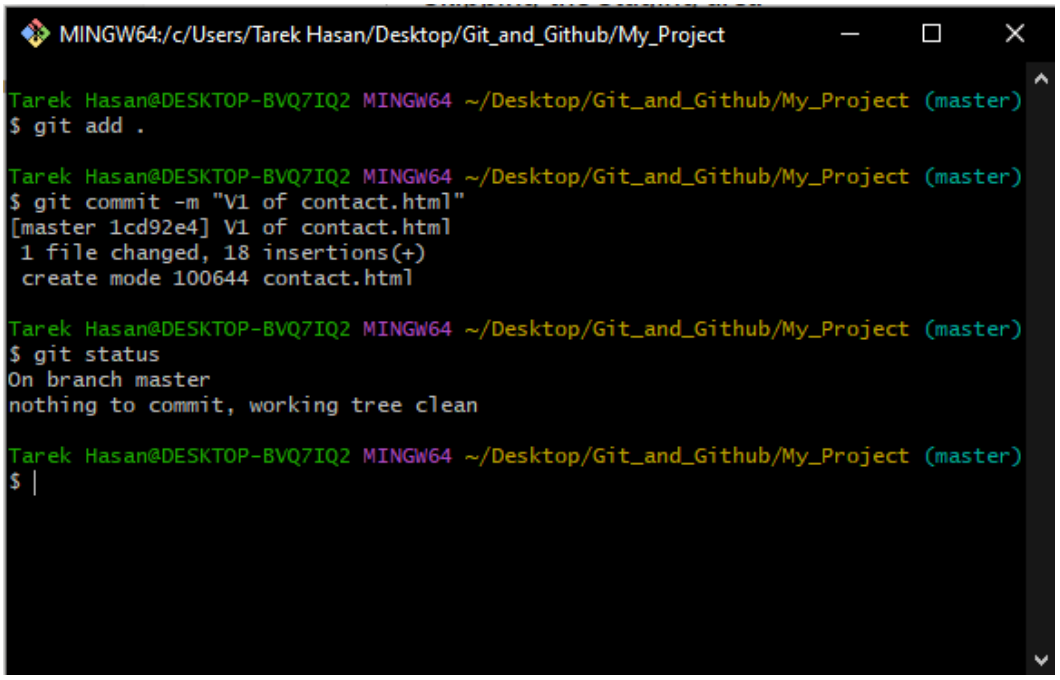
Here we have unstaged the file.

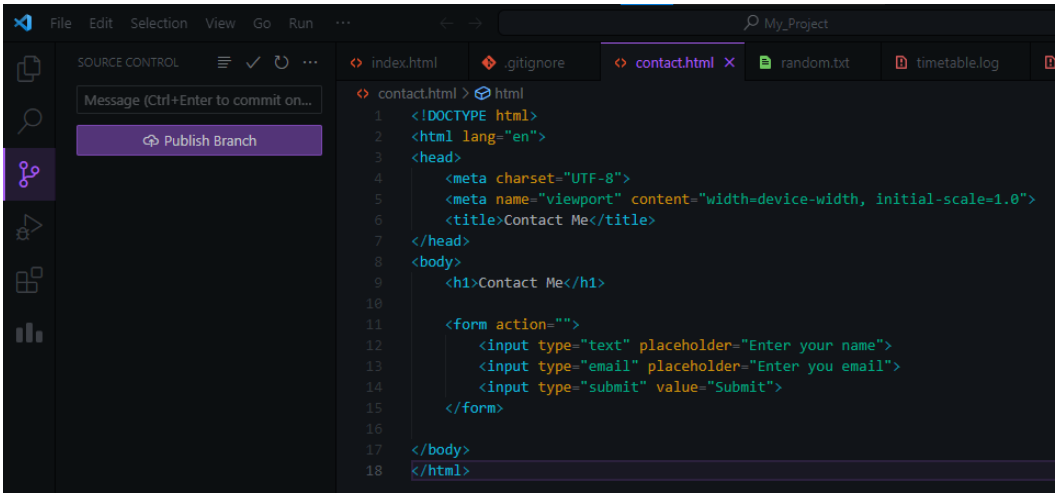
▼ **Unmodifying Files [Going to the previous version] [Uncommitted Files]**

▶️ Unmodifying in here means going back to the previous something. Suppose we have written some codes in the contact.html file. But now the new code has some faults and we want to go back to the old code. We can do this by running:

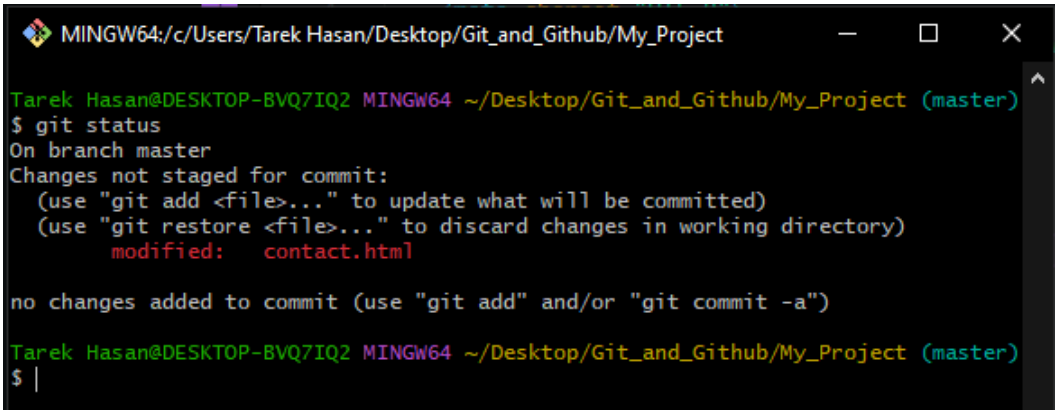
> 📌 git checkout — <FILENAME>.extension

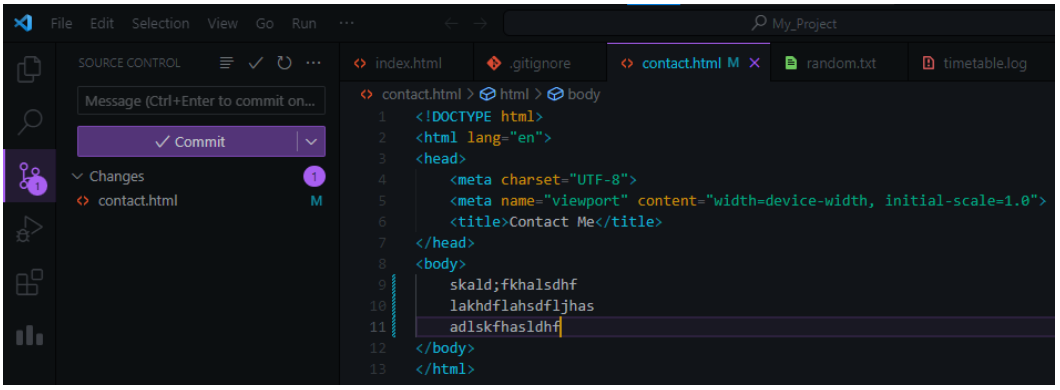
This will take back your file to the previous version. Let’s look at the practical example. First we are going to commit and will look into the VS Code window



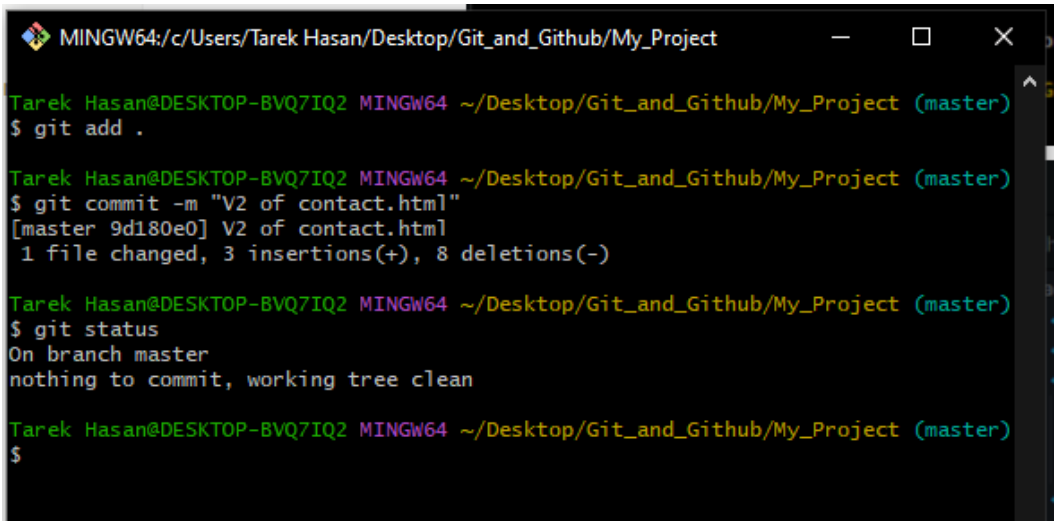


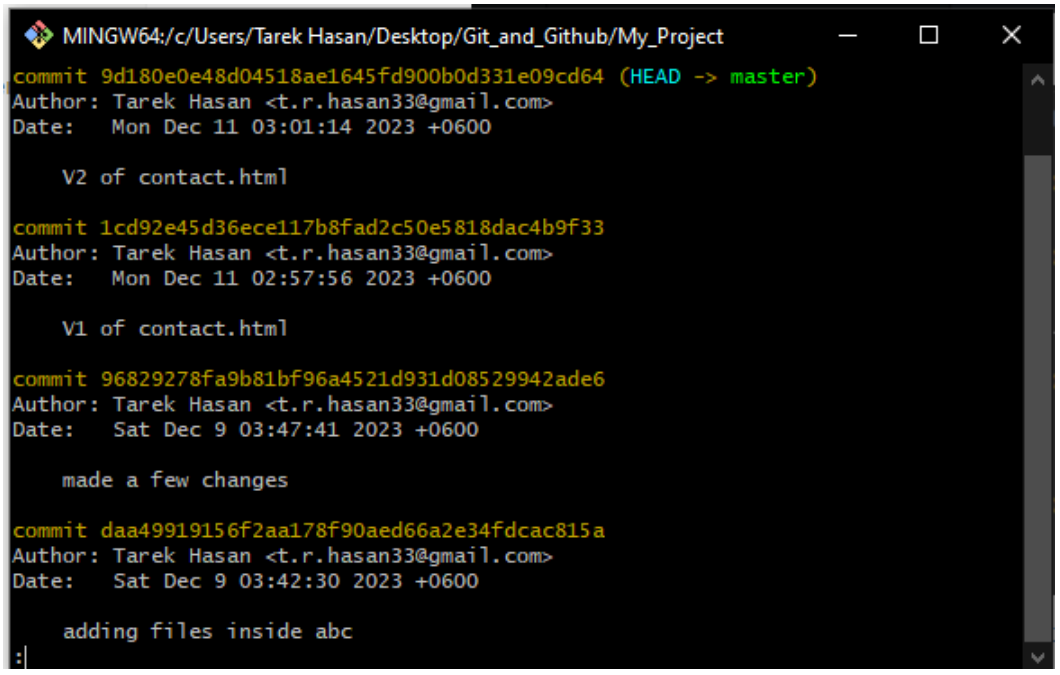
Now we have committed the file. We have the version 1 of contact.html. Let's remove the body part of the contact.html and write something odd:



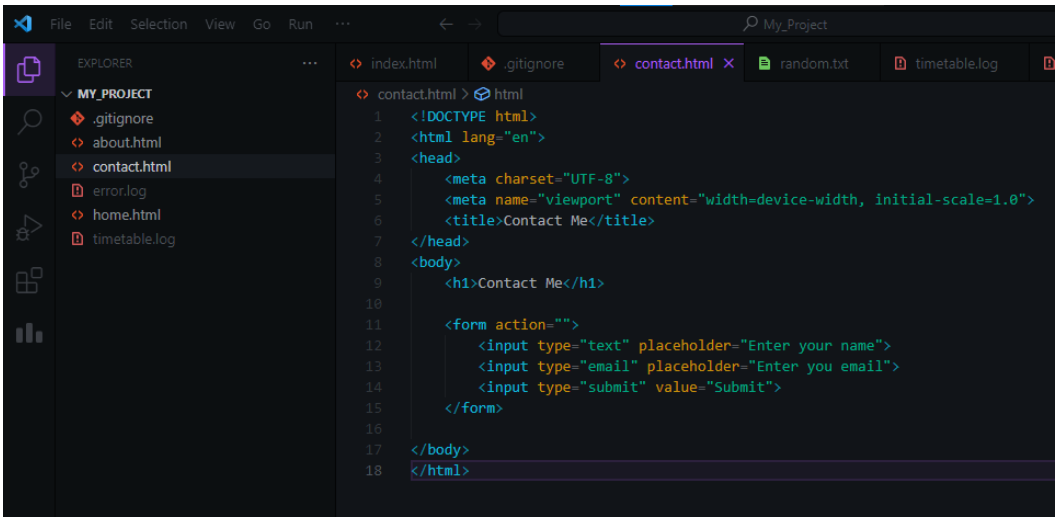


But now we don't like this code because it isn't functioning. To go back to the previous code we can run the command:





Here it is. Now let’s go back to the previous version of the file and look at the git log as well as the VS Code window:



Here it went back to the previous version. Suppose you have made some changes to the every file and the project is not working now. You can go back to the previous commit by running this command:

> 📌 git checkout -f

▼ Git Alias

▶️ What does Git Alias means? It means name of something. Let’s think about the git status command. If you think that the git status command is too long you can make it short by running a command like this:

> 📌 git config —global alias.<SHORTNAME> <COMMAND NAME or LONG NAME>

Suppose you want to set status as st so it will be shorter:

> 📌 git config —global alias.st status

Now if you run git st it will show you the status. Let’s think about the unstage/untrack command. Here is the command:

📌 git restore —staged <FILENAME>.extension

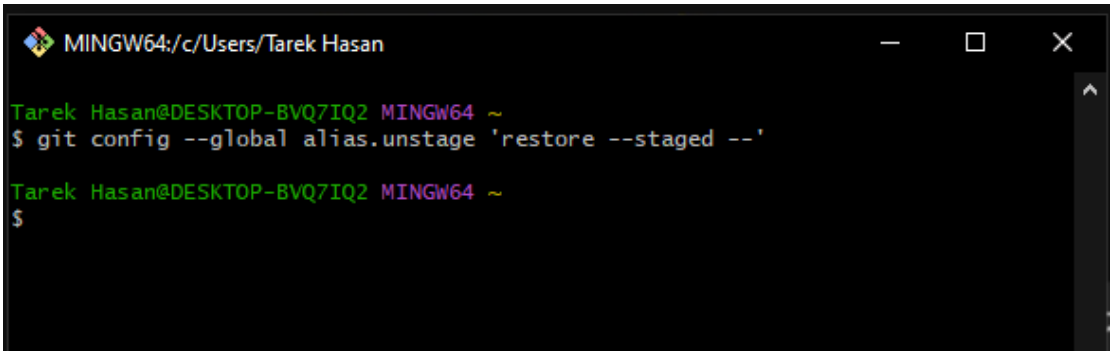
We can make it short by using the command:

📌 git config —globar alias.unstage ‘restore —staged —’

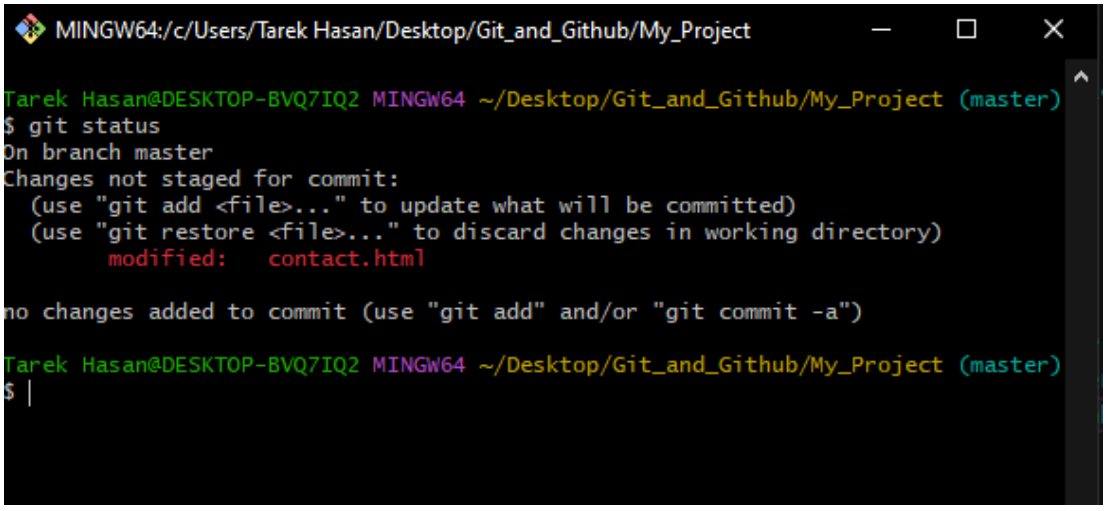
We can run the command by:

📌 git unstage <FILENAME>.extension

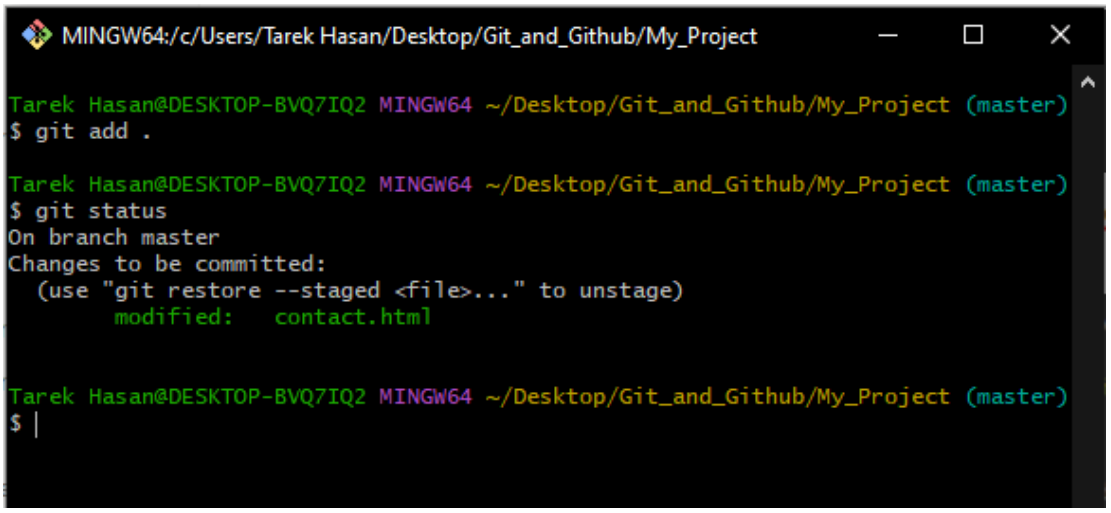
So we want to make it a short. Let’s see the git bash window:



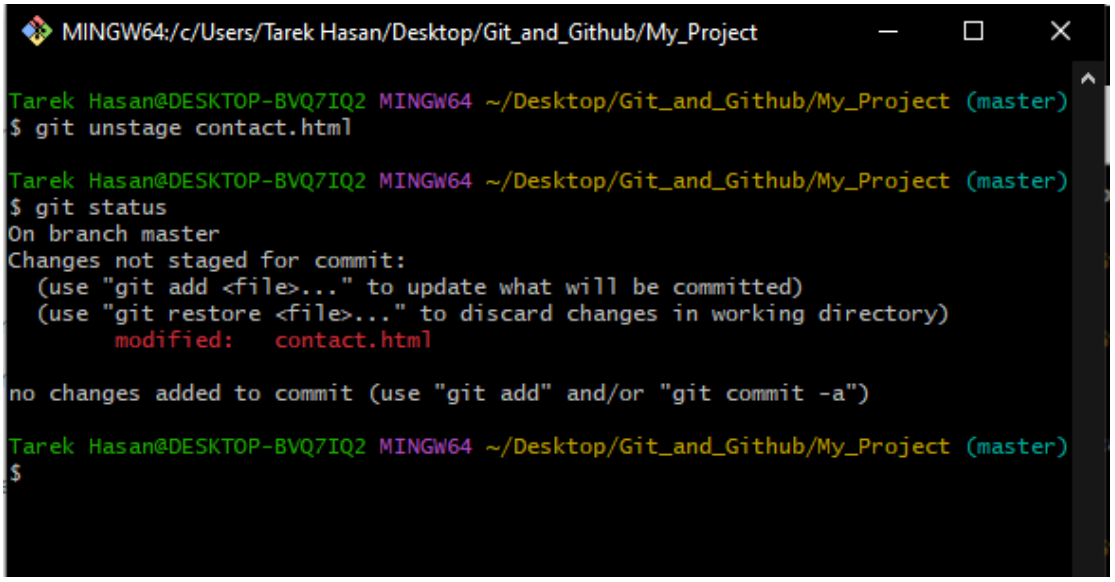
Now we can just use the command git unstage and it will do the work. Let’s modify something in contact.html and see the git status:



Let’s stage the file and see the status:



Let's run our short command and look into the status again:



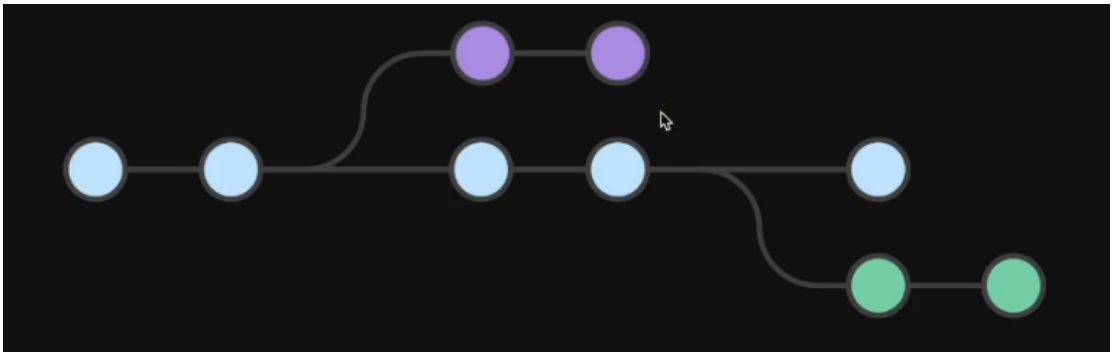
It has been unstaged.

▼ All About Branching

▼ Branching

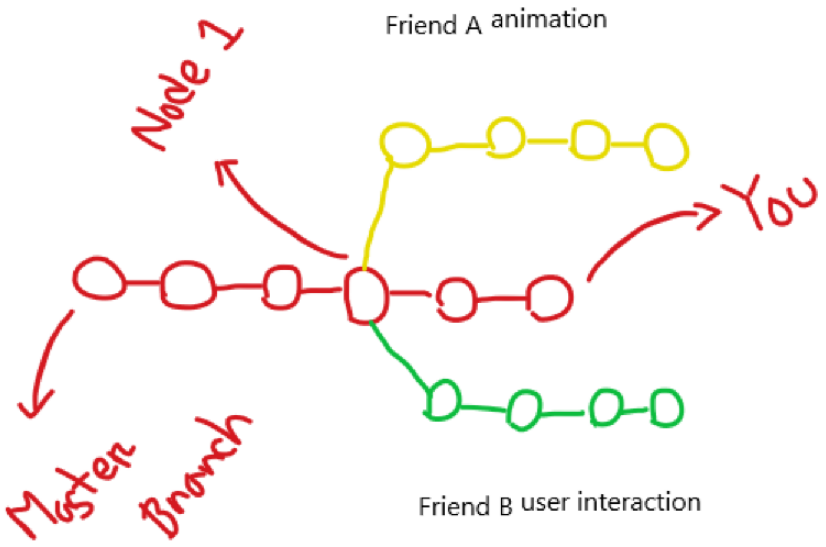
▶️ In version control system, Branching is one of the most important feature. We have a default branch name MASTER as you may have already saw this in the git bash window. It is our project branch. It contains all the commit that we have made. Suppose you want to add a new feature to the project. Git gives you the opportunity to add a new feature to your project without changing the code of the whole project. What it means is basically you are creating a branch beside the main project. And if you don't like that branch you can delete it.

⭐ **Branching:** Diverging from the main branch.



▼ **Why Branching?**

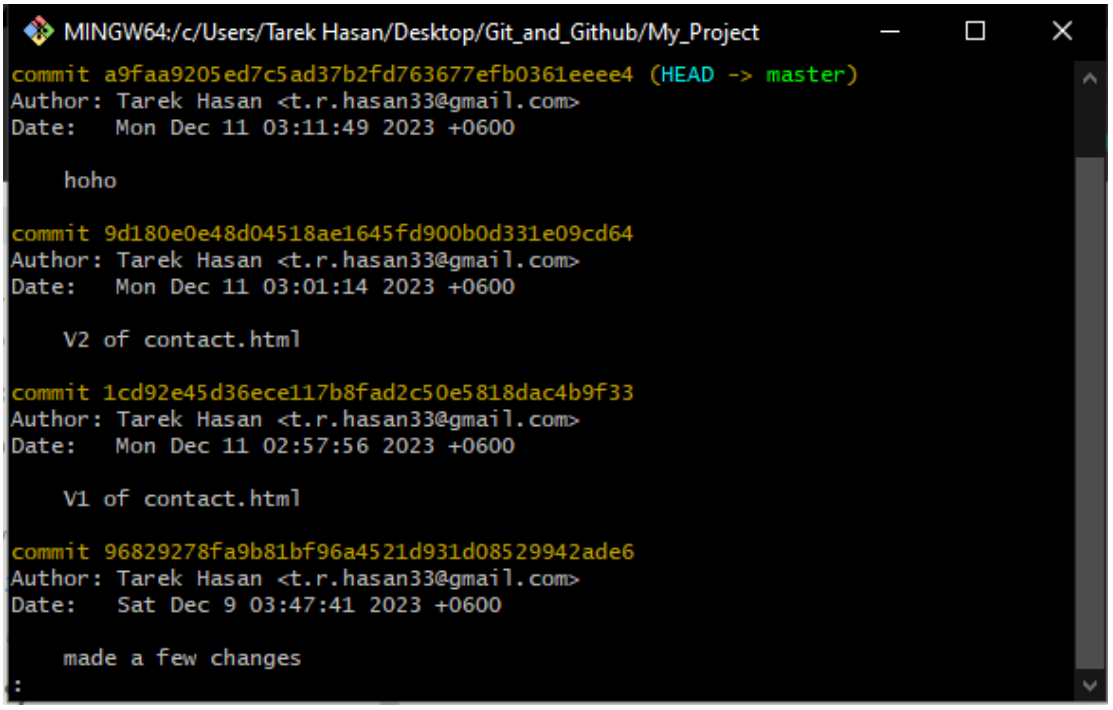
▶️ Why do we need branching? Look at the picture below:



Here, suppose you're the creator of the master branch. Till Node 1 you have committed many things and you project is working. But then you Friend A and Friend B wants to contribute to the project with animation and user interaction feature. But is it the right way to give them the whole project code to work one? If something wrong happens it will effect the whole project. So you let them create their own branches from Node 1. They have the access of all the commits till Node 1 as well. They can now work on the project on their own and you will also be able to work on your project on the master branch as well. But they won't have the access to the commits after Node 1. And if anything doesn't work for any branch you can just delete the branch. If a branch works very well you can marge the branch in your master branch.

▼ Creating a Branch

▶️ At first we need to know about what is HEAD. We already have our project going on. It's still on the master branch and the last commit is what we call HEAD. Let's look at the git log.



Here is shows the last commit as HEAD and it's on the master branch.

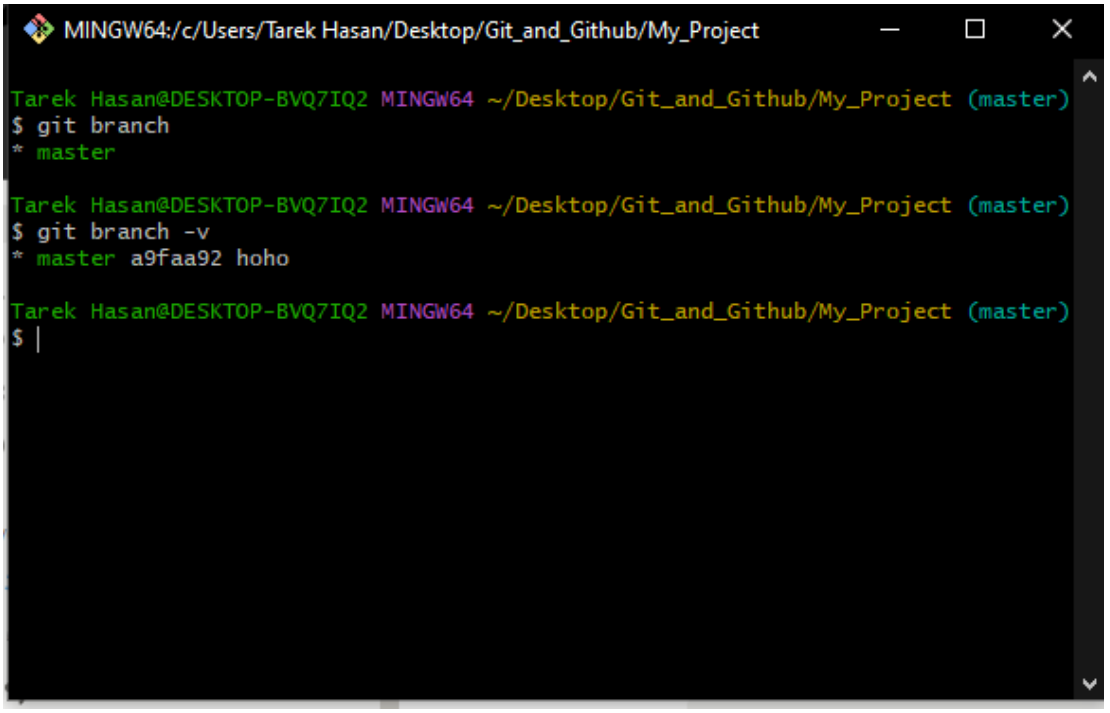
Here are some commands:

📌 git branch

This shows all the branches you have.

📌 git branch -v

This shows the branches with details like latest commit hash and message. Let's look into the practical example:



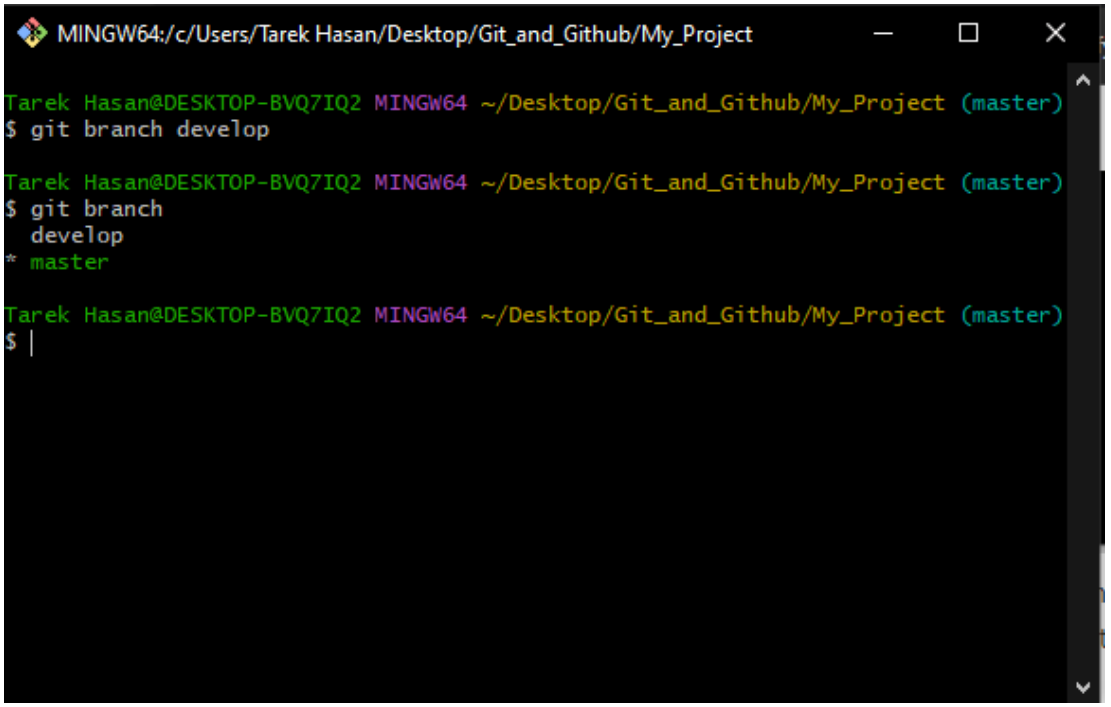
Here you can see the first command shows in which branch we are currently working using the * mark. The second command shows the branch with latest committed hash and message.

▶️ Let's create a new branch:

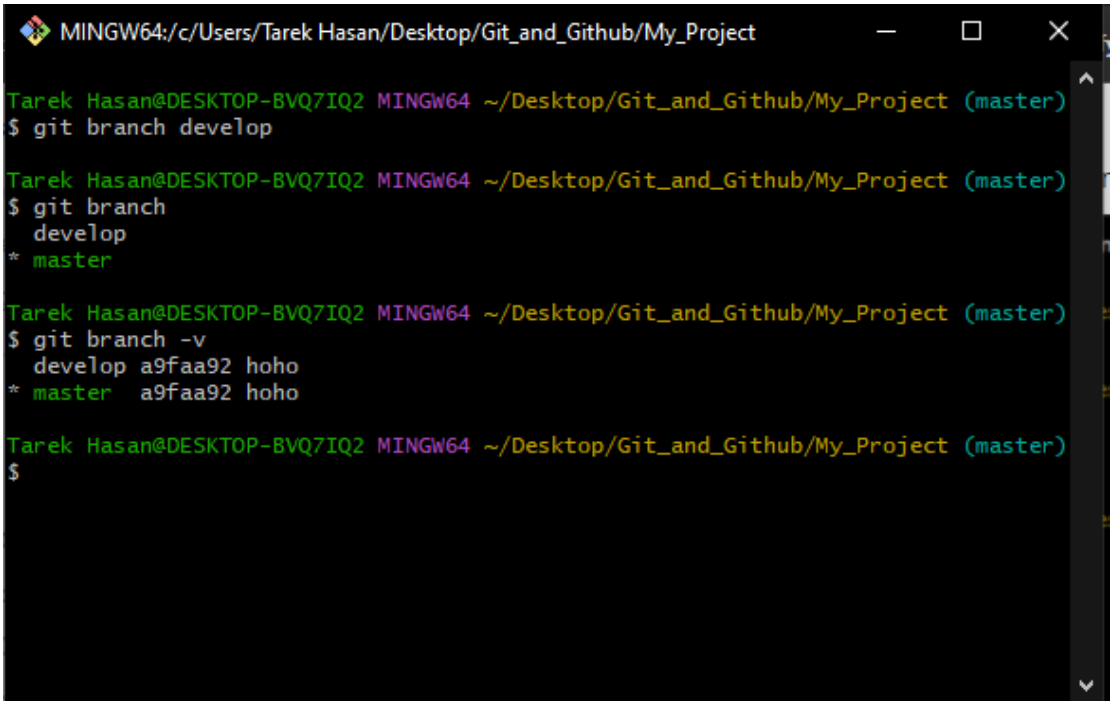
To create a branch we can use:

> 📌 git branch <BRANCHNAME>

Let's create a branch and look at the branch status:



So we have created a new branch named develop and it also shows that we are currently in the master branch. Let's run the other command:



They both shows the same hash value and message right? Why is that? Cause we have created a branch in the latest commit. The branch develop will have access to all the previous commit till this commit. But it won't have access to the future commit in this master branch.

▼ Switching Branches

▶️ We have created a branch named develop earlier. We will try to switch to that branch now. To switch between branch we can use:

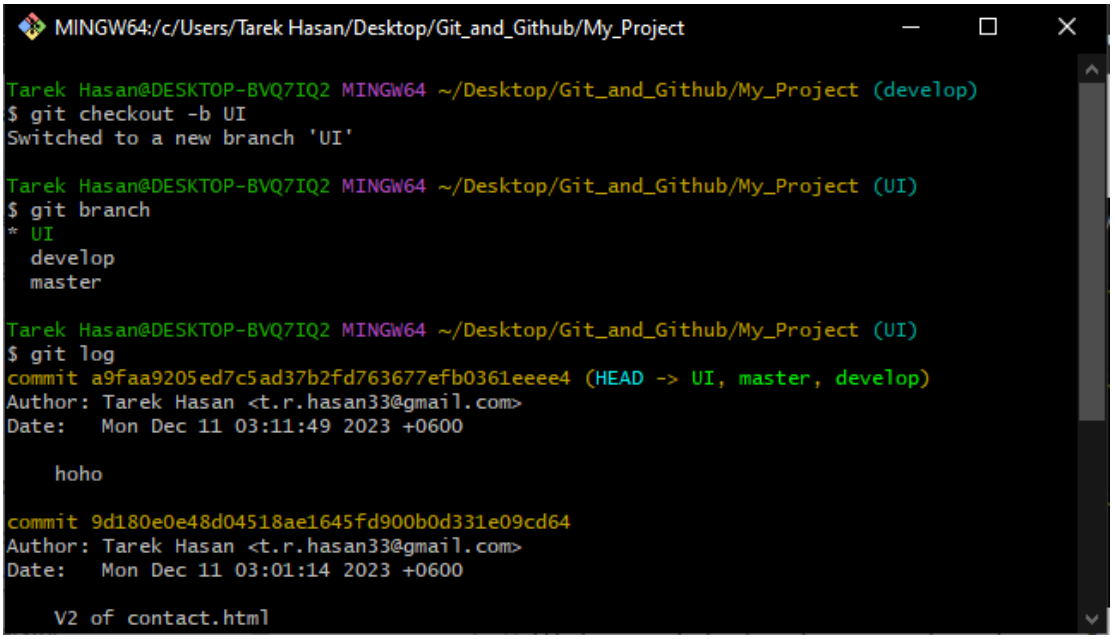
> 📌 git checkout <BRANCHNAME>

Let’s look into the practical example:

We have switched to the master branch now. Suppose, now you want to create a branch and switch to that immediately. We can do that by using:

> 📌 git checkout -b <BRANCHNAME>

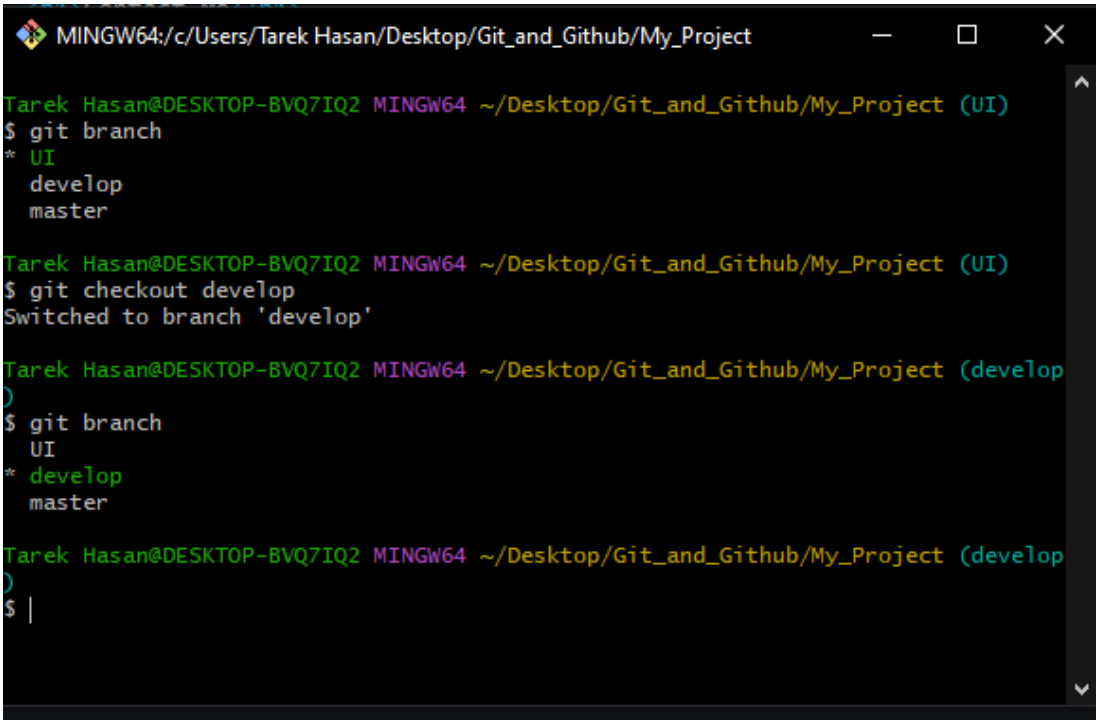
Let’s look into the example:



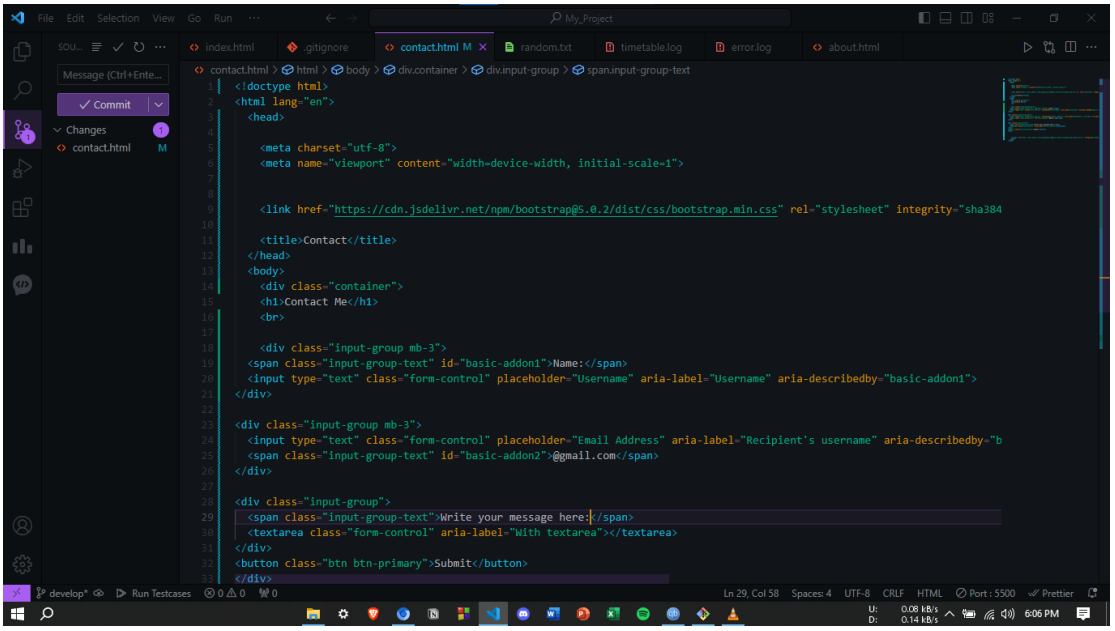
Here we have created and switched to the branch at the same time. The git log also shows that we are in UI branch.

▼ Working with Branching

▶️ Till now we have known about branching and how to create branches and all. But now we will look into the functionality of branching. We have already created 2 new branches along with the master branch which are develop and UI. So what happens if we create a file in develop branch? As we know the develop branch has access to all the commit till the commit where we created the branch. So what happens when we change something in the develop branch and we get back to master branch? Will our codes in master branch change too? The simple answer is “What happens in another branch stays in that branch”. So nothing will change in the master branch. Even if we create a new file in develop branch it will not show up in the develop branch. Let’s look into the practical examples:



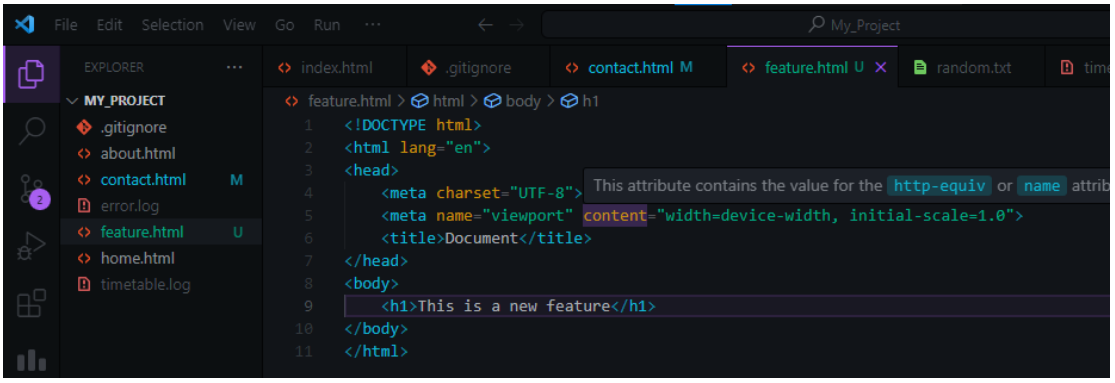
We have switched to the develop branch. Now we will edit the contact.html file.



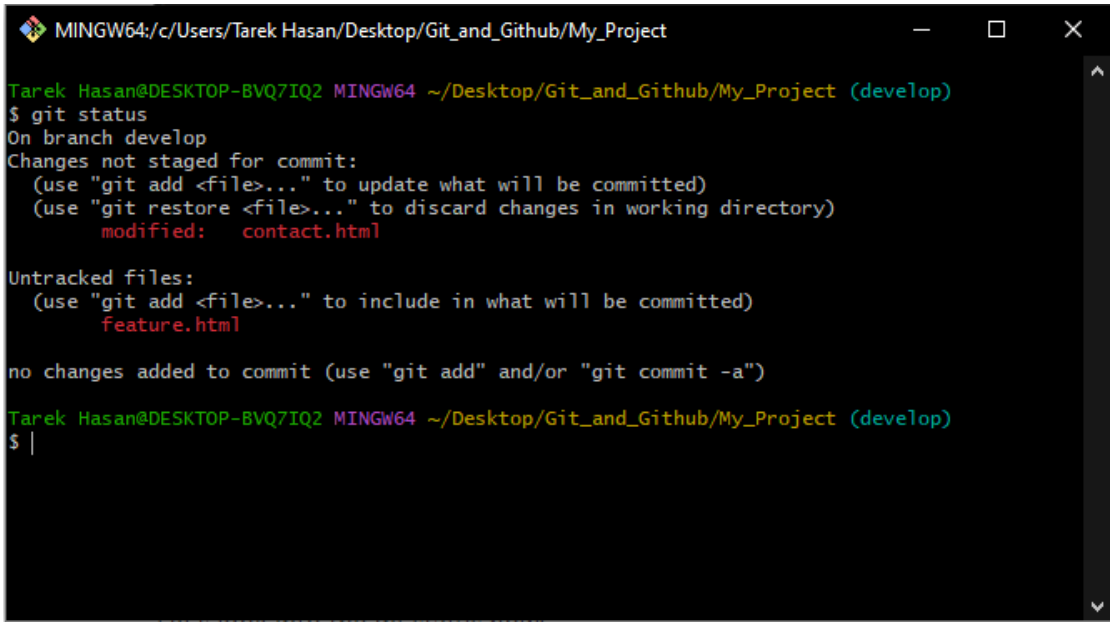
This is our whole code in contact.html in the develop branch. Now let's check the git status:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (develop
)
$ git status
On branch develop
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   contact.html

no changes added to commit (use "git add" and/or "git commit -a")

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (develop)
$
```

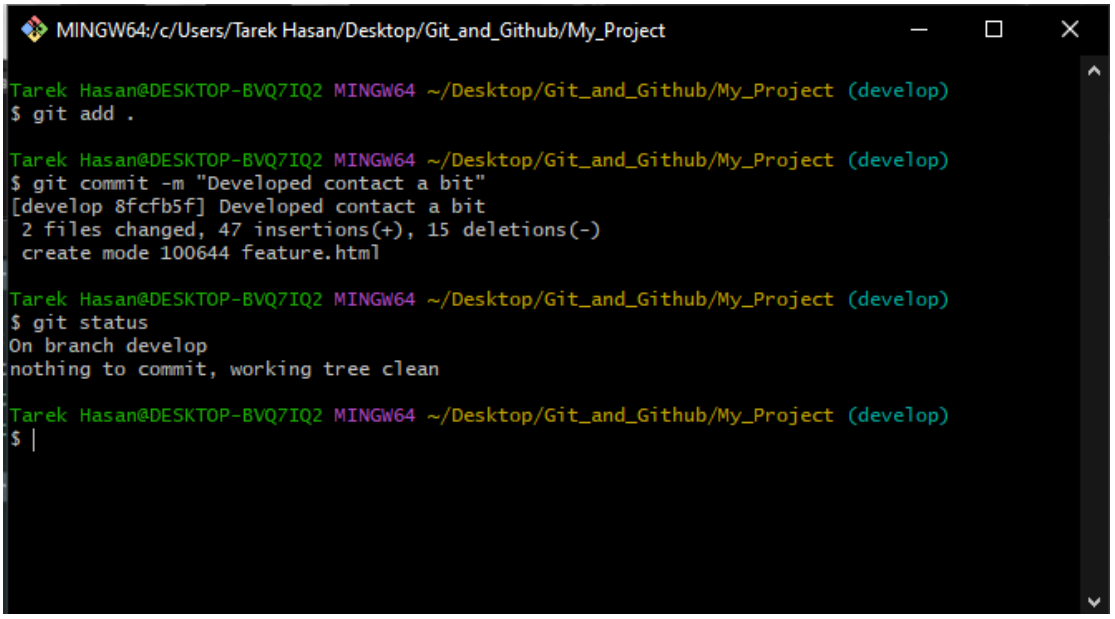
Let’s create a file in the develop branch named feature.html and add basic html code in it.



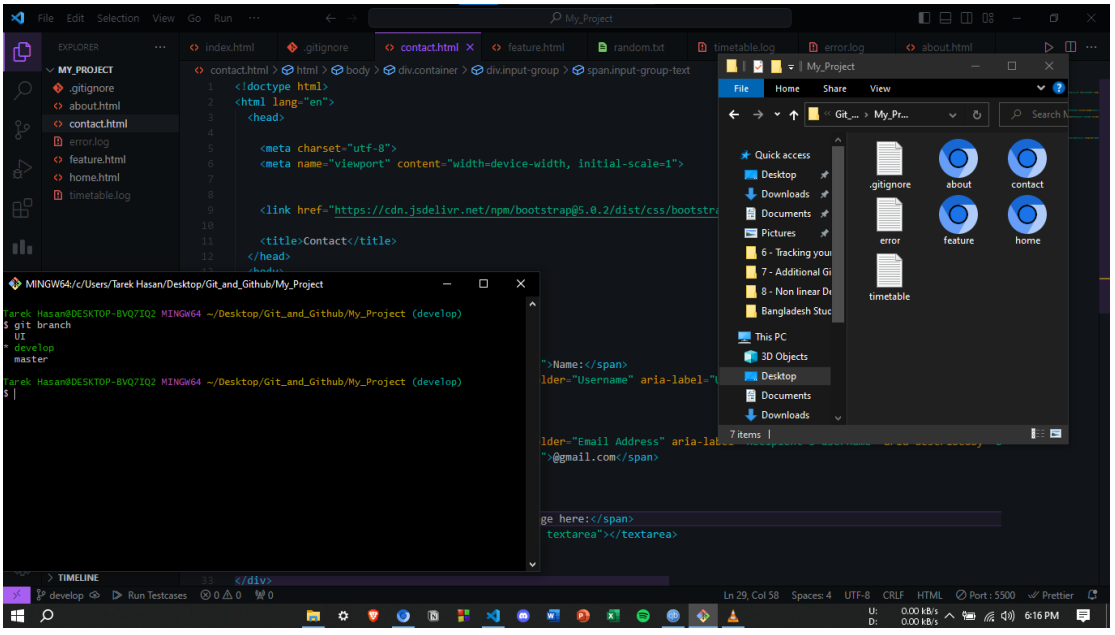
Let’s look into the git status now:



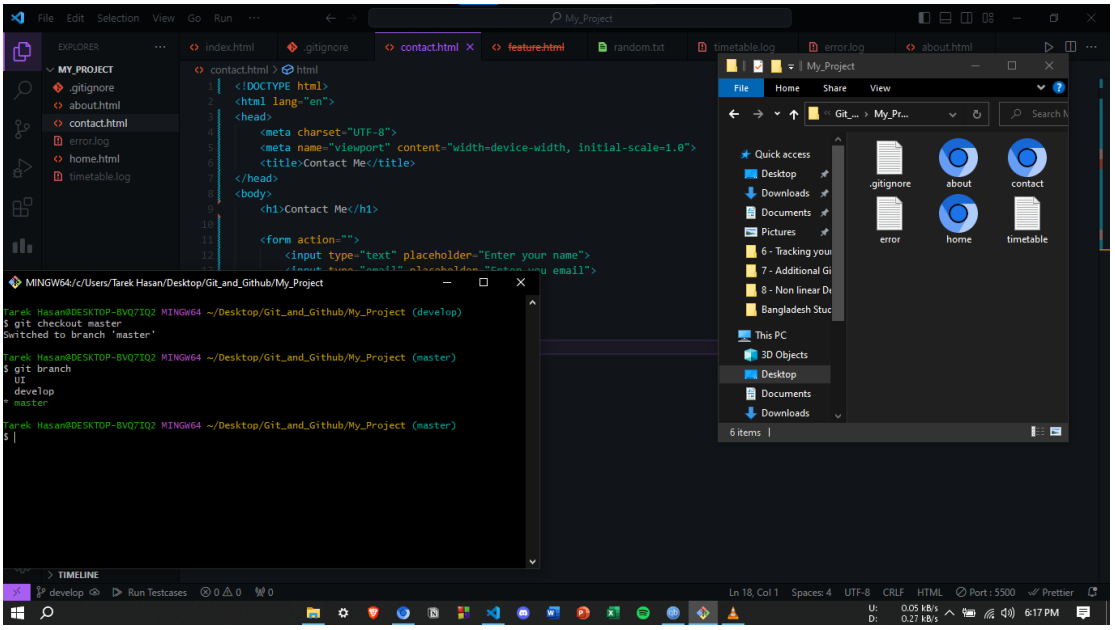
Let’s add them to staging area and commit them then we will look into the git status:



As we know we are currently in the develop branch…let just look into the changes all together in a windows. Then we will switch to the master branch and then we will get to know the functionality:



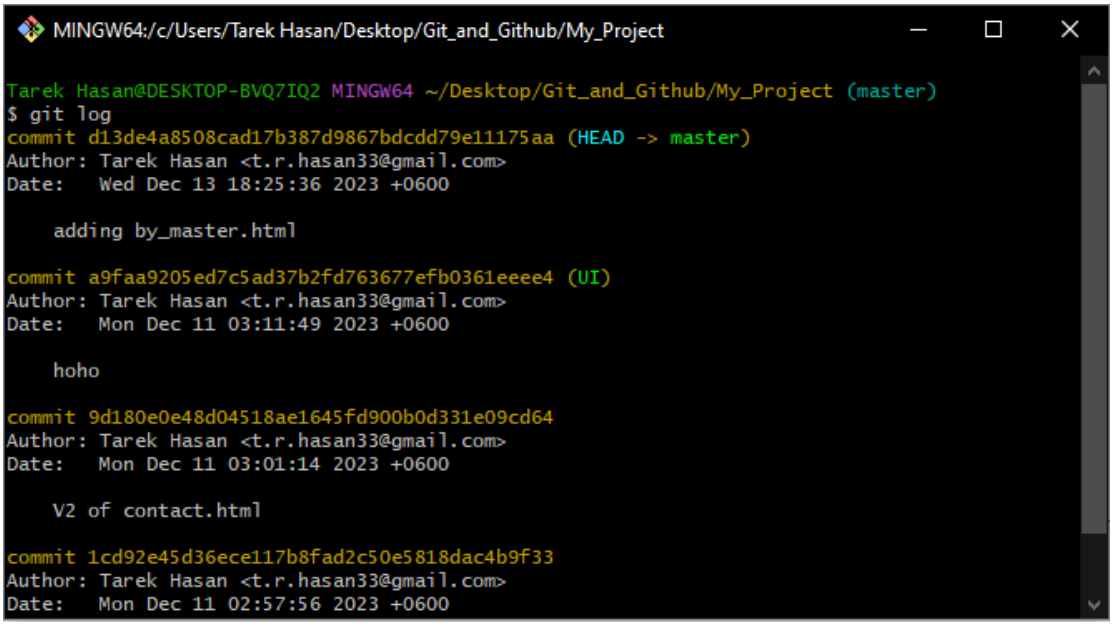
We are in the develop branch here…Look at all the window. We have feature.html and the modified contact.html as well. Let's switch the branch and look into the window again:



Look now. The feature.html file is gone. The code has changed. If we now add a file in master branch it will not show up in the develop branch. The HEAD will shift to that branch which is modified the last time.

▼ Git logging

▶️ We already know about git log right? It shows information of all the commits that we have made. We are currently in master branch. Let's look into the git log.



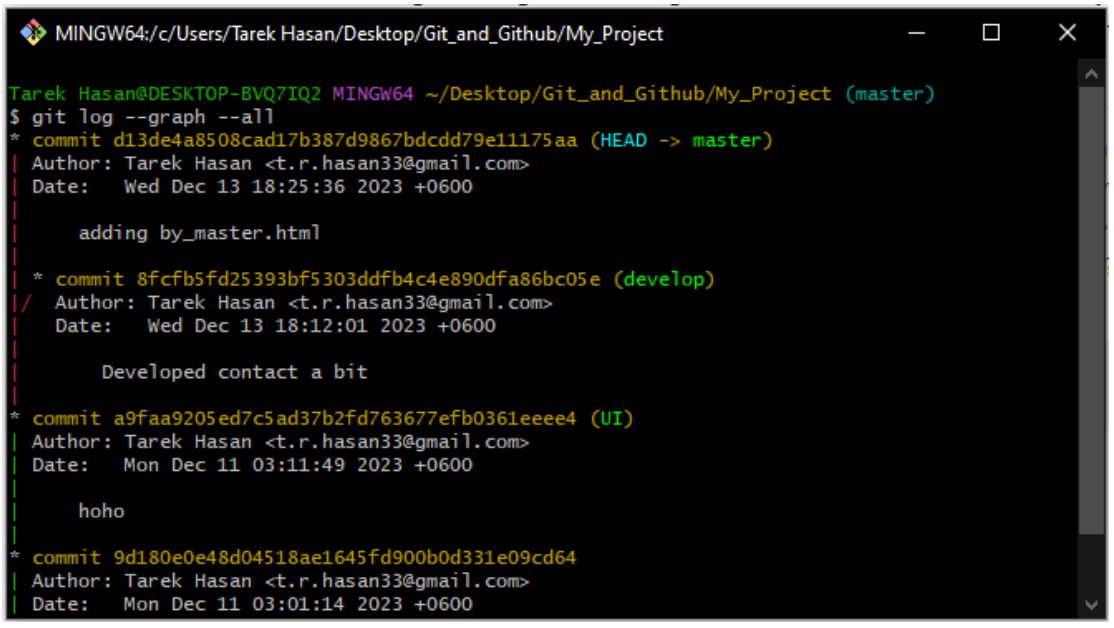
What is the last log showing. It's showing that we have added a file named by_master.html. But shouldn't it be showing the changed we made in develop

branch earlier as well? Why isn't it showing this? Also look at the 2nd log. It shows the UI branch. Let's explain.

▶️ We had created 2 branch, develop and UI along with master branch. All had the same commit till contact.html. Then we have modified the contact.html in develop branch and then we have added another file in master branch by switching to master branch. So develop and master branch went in different path from the last commit, and UI is still on that commit. But another question is why isn't the develop branch is showing in the git log? Cause we are in the master branch. But it's possible to look into the whole log by running this command:

> 📌 git log —graph —all

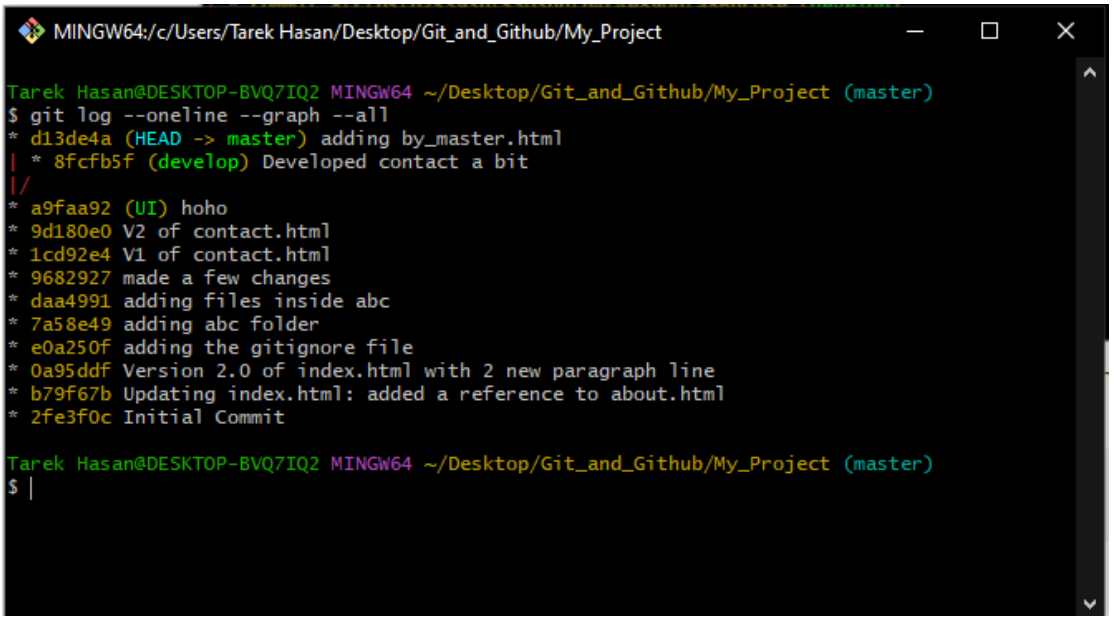
Let's look into the example:



Now it shows the commit of develop branch which we did earlier than the commit in master branch. To look into the commit in one single line we can use:

> 📌 git log —oneline —graph —all

Let's look into it:



▼ Deleting Branch

▶️ We have 2 branches as of now, which we created later. We had a master branch previously. The work in the develop branch is done by another person, and the UI branch hasn't been touched since then. This means that the UI branch hasn't been edited. Now we want to delete the develop branch. But git will not let us delete this branch. Why? Cause we haven't merged it yet. So if we delete that branch all the development will go away. But we can delete it using another command. Here is all the commands to delete branches:

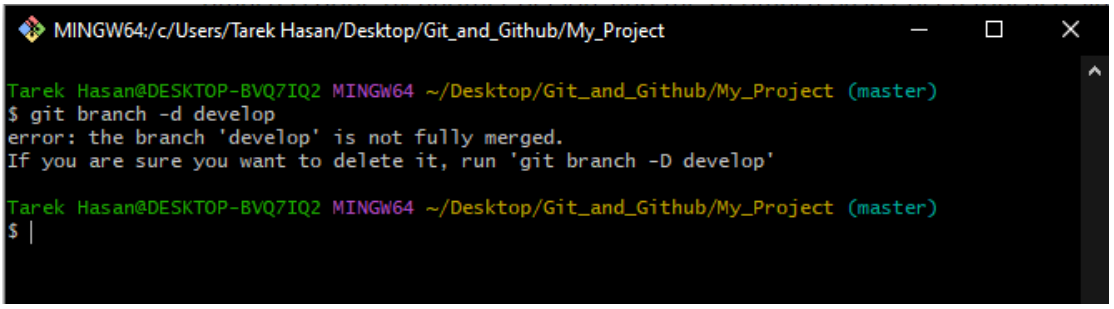
To delete merged branch:

> 📌 git branch -d <BRANCHNAME>

To delete unmerged branch:

> 📌 git branch -D <BRANCHNAME>

let's look into the example:



As you can see it isn't letting us delete the develop branch. Let's try with the UI branch:

It has been deleted. Why? Cause we didn't modify any files in the UI branch. We can also delete the develop branch as well using the other command.

▼ All About Merging

▼ Merging

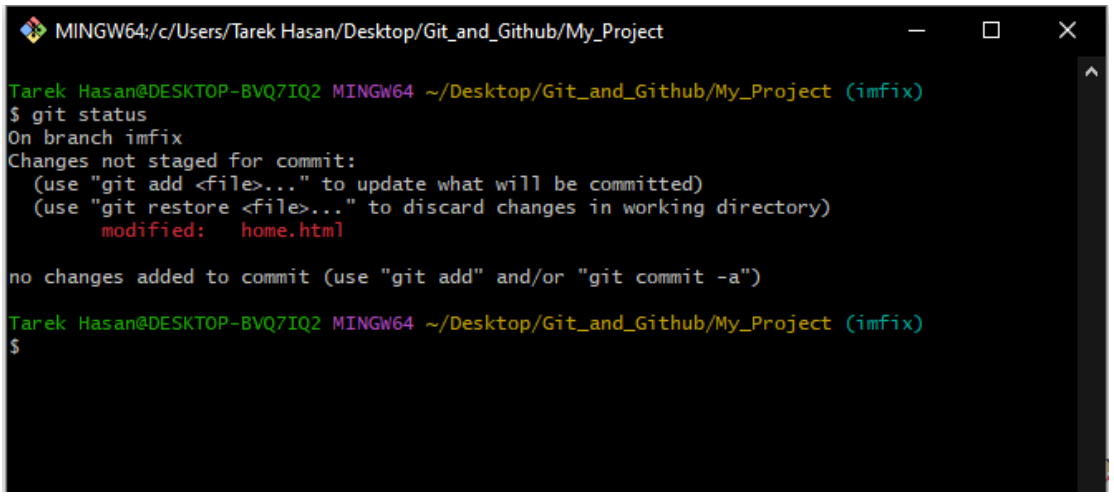
▶️ What is merging actually? Why do we need this? Merging means mixing the later created branch with the master branch. Why do we need merging? Suppose you have your project. Now you want to modify the index.html file and add some new feature. But doing this on your master branch and changing the main source file? Wouldn't it be a little bit risky? That is why we crate a new branch and modify everything on that branch and if we think that the development in new branch works well we can merge it into the master branch.

▼ Basic Merging

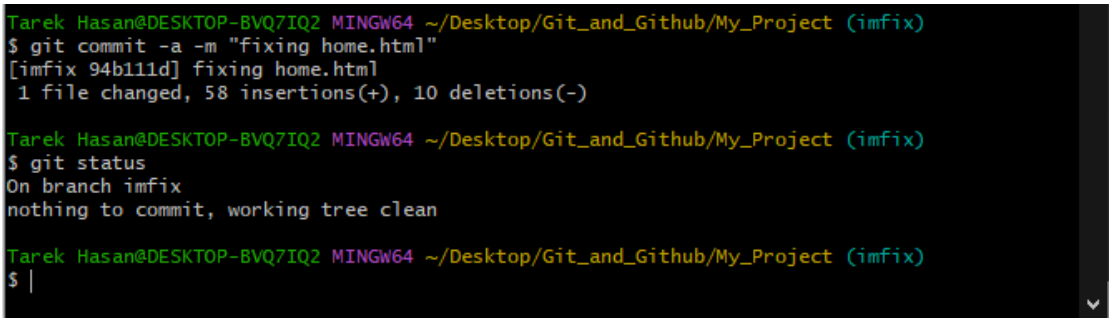
▶️ We already have develop branch. Now let's create another branch called imfix branch using the command:

> 📌 git checkout -b imfix

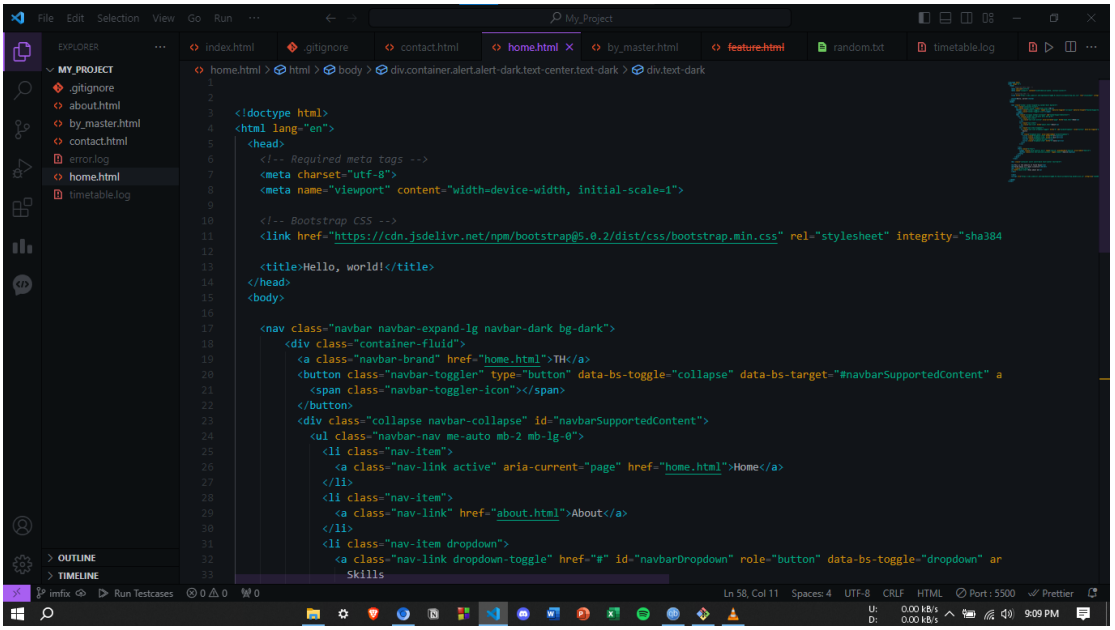
After that suppose we have done some changes to the home.html to make the home page more interesting. Now let's see the git status:



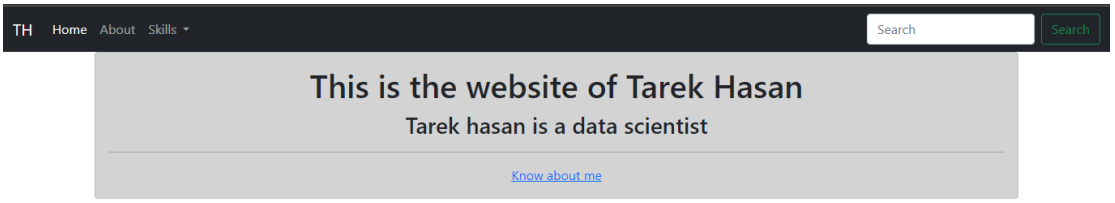
Let's stage and commit the file at the same time:



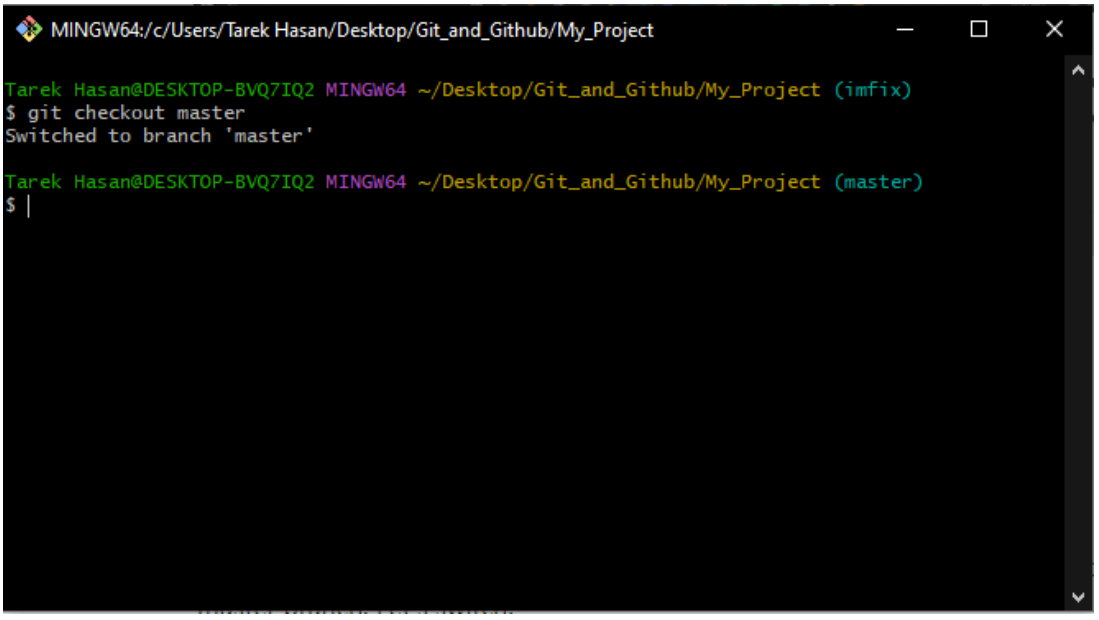
Let's see the VS Code window for the home.html of master branch:



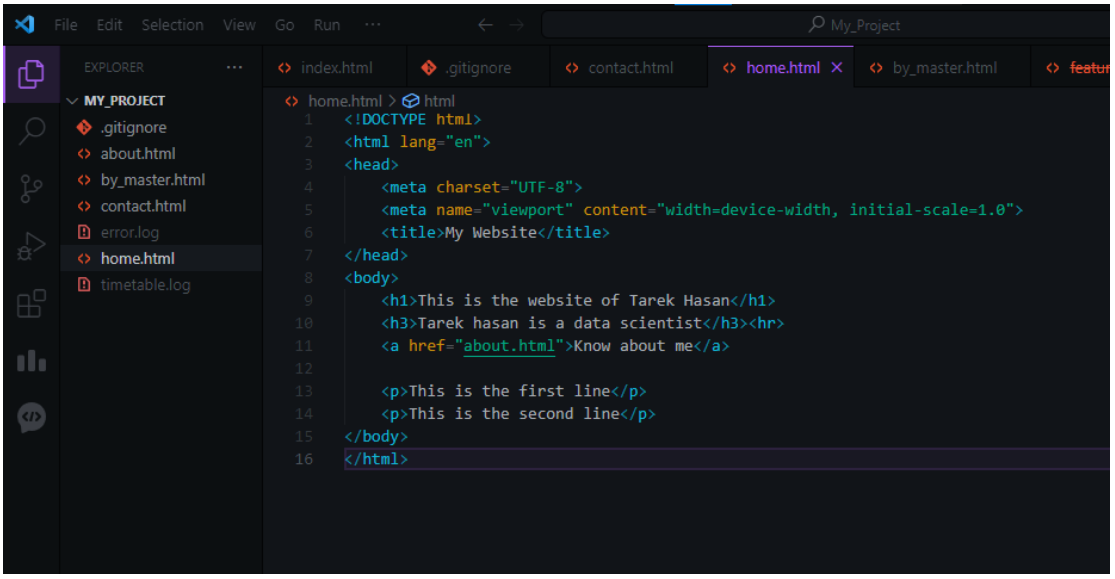
Let’s see the browser window:

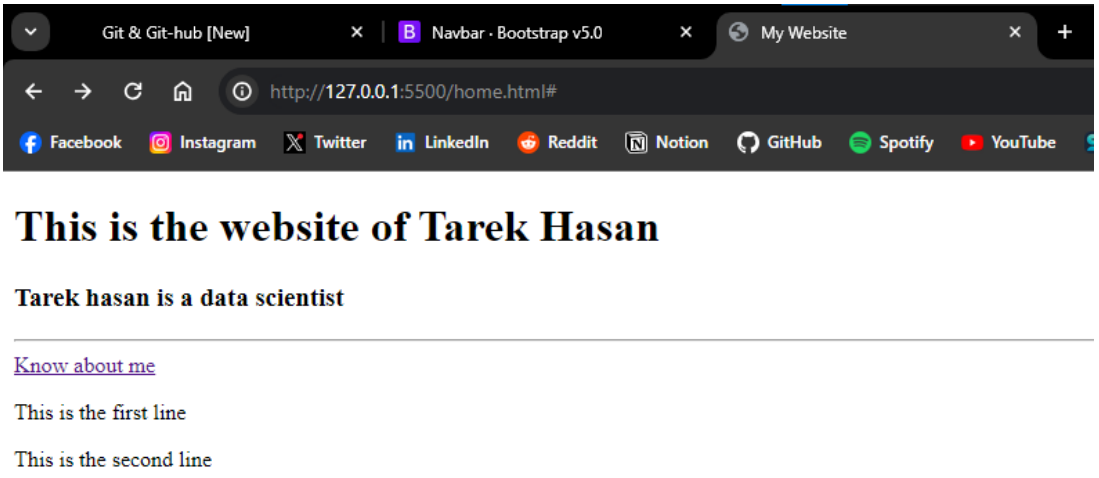


Now we want to merge it to the master branch. We are currently in the imfix branch. To do this we need to switch to the master branch. Let’s switch:



Let’s look at the VS Code window and browser window for the master branch:

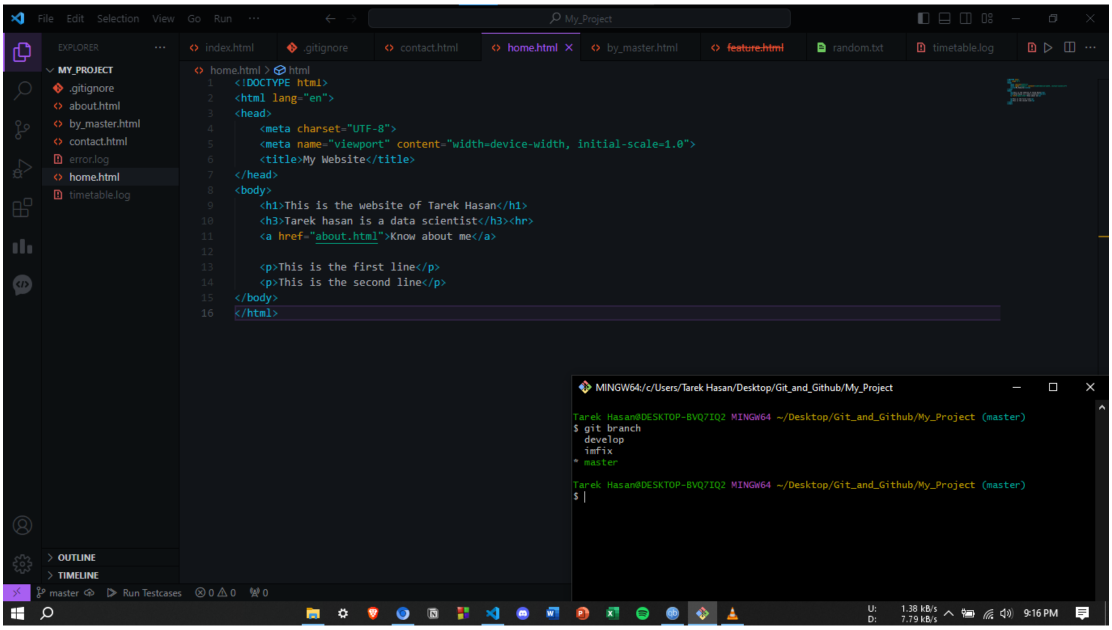




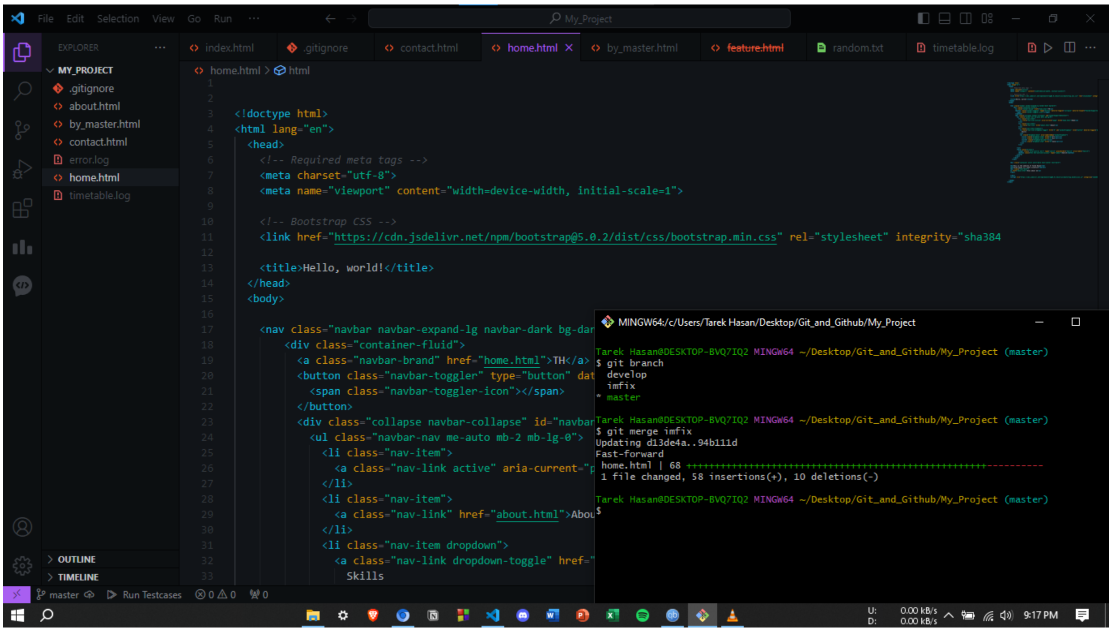
▶️ Let's begin with the merging. To merge a branch we can use this command when we are in the branch with which we want to merge another branch. In this case we are in the master branch and we want to merge imfix branch with master branch:

> 📌 git merge <BRANCHNAME> [In this case the branch name should be imfix]

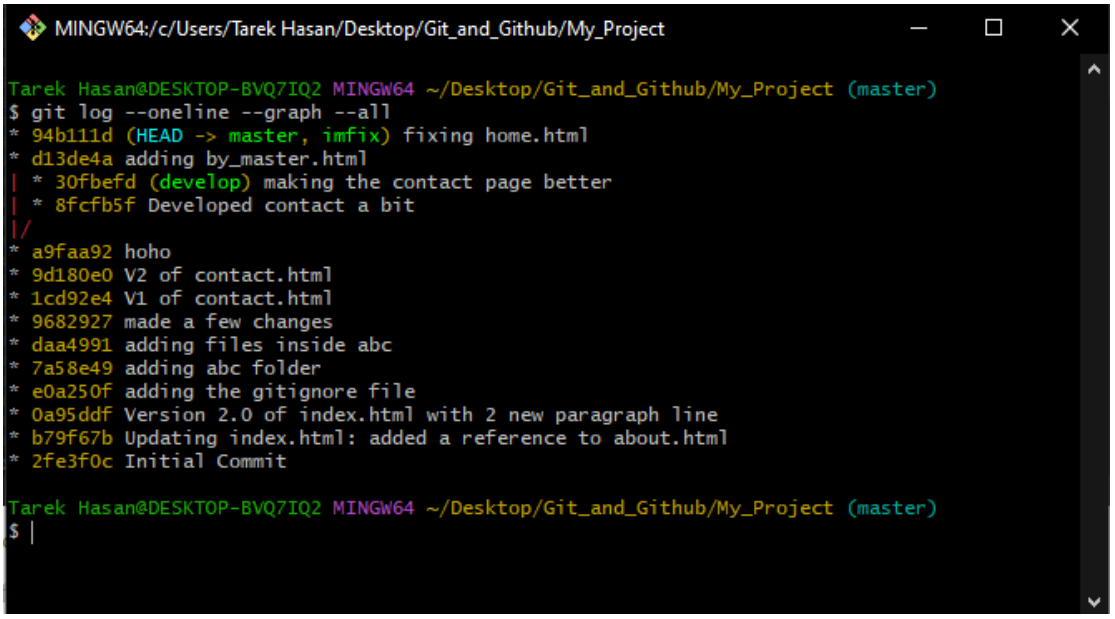
Look at this picture:



We are currently in the master branch and the home.html code is there. Let's run the command and look into the window again:



Everything has changed. Also look at the git bash window. Let's also look into the git log:

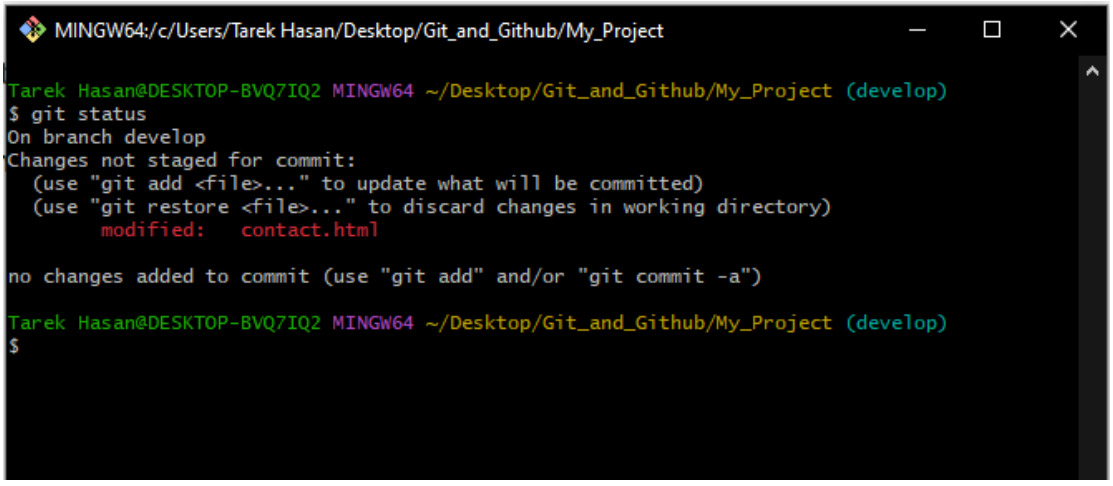


The branches has been merged.

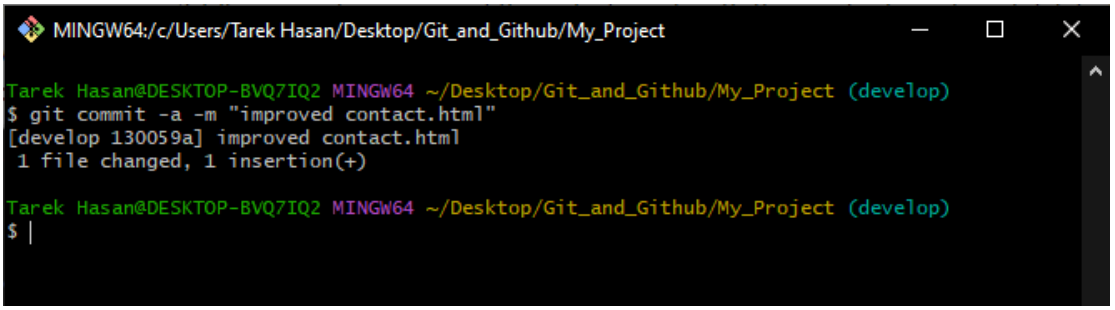
▼ Recursive Merging

▶️ What is actually recursive merging? Till now we have merged the imfix branch with the master branch and deleted imfix branch. Now we want to make some changes into the develop branch. Suppose we have made some changes. So the basic merging technic is we switch to the master branch and merge the develop branch. But we do we call is recursive merging? Cause we have already changed so many things in the master branch. We have changed the home.html in the master branch. But the home.html in the develop branch is still the old one. So while we merge the develop branch now won't it also change the home.html which we changed earlier to the old version because develop branch has the old version? It won't because of recursive merging. Recursive merging checks the source node of the branching. Then it checks the latest commit and changes to the branch we want to merge. It will not touch home.html because home.html in the develop branch is as same as the source commit. So it will just modify contact.html.

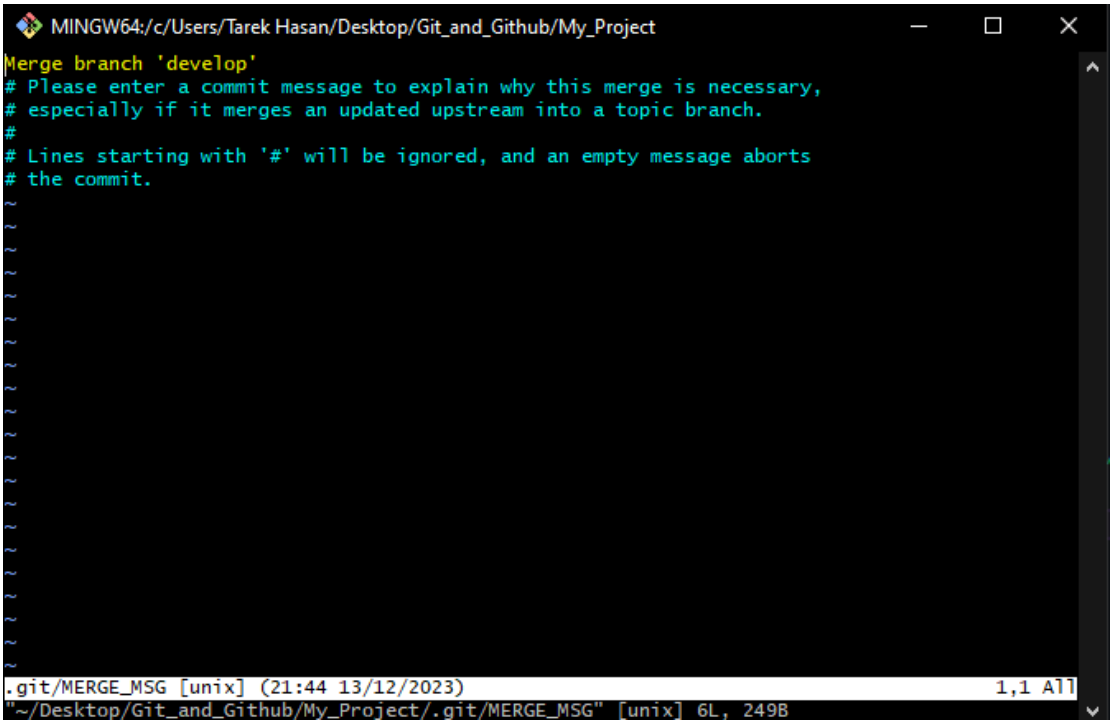
We have modified the contact file. Let's look into the git status:



Now let's stage and commit the file:



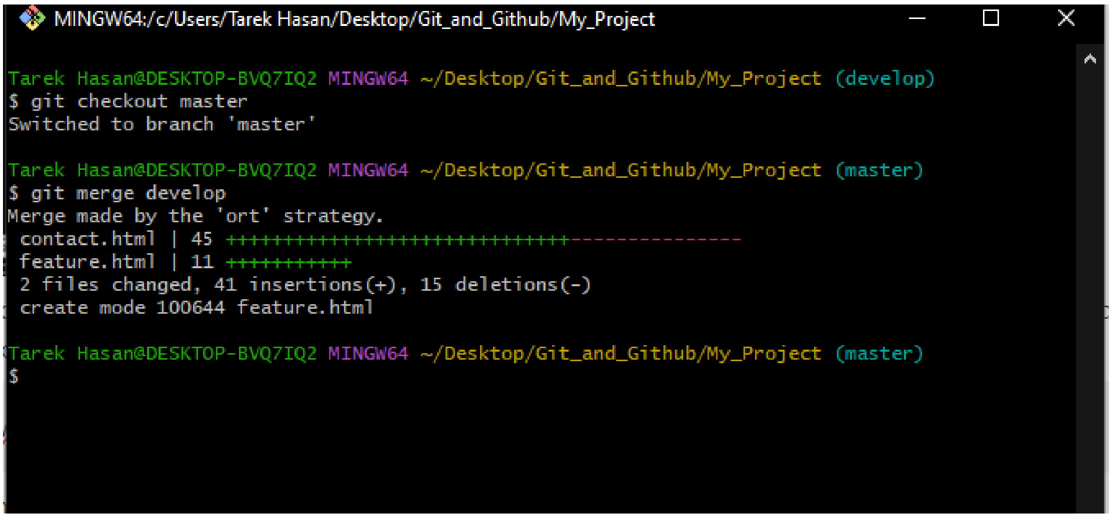
Now let's switch to the master branch and try to merge:



It will show you a message like this. Here the yellow line the commit message. Ignore the other line. How to complete the task now? Look:

> 📌 press **:wq** and then enter

Then it will show this window:

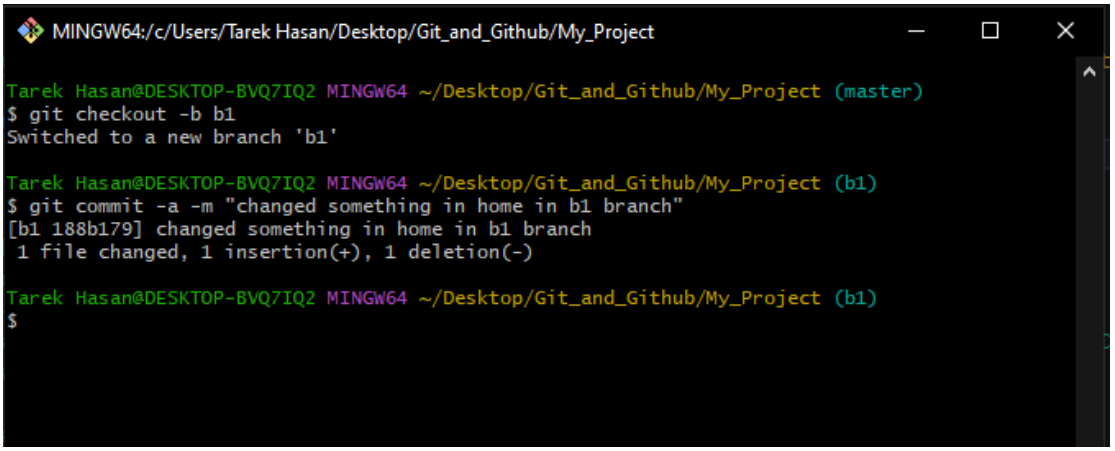


The branch has been merged.

▼ **Conflicts in Merging**

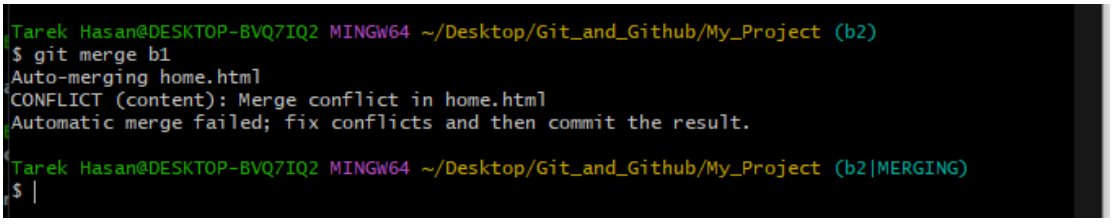
▶️ Why merging conflict happens? Suppose we have created 2 new branches of master named b1 and b2. Now we have changed a line in home.html in b1 and then switched to b2 and changed the exact same line in b2. But I want to merge b1 into b2. It will result as an conflict because we have changed the exact same line in b1 and b2. Git can't decide which one to keep. Let's see the practical example:

First let's create b1 and change something then commit:



Now switch to the master branch and create b2 branch and do the same thing:

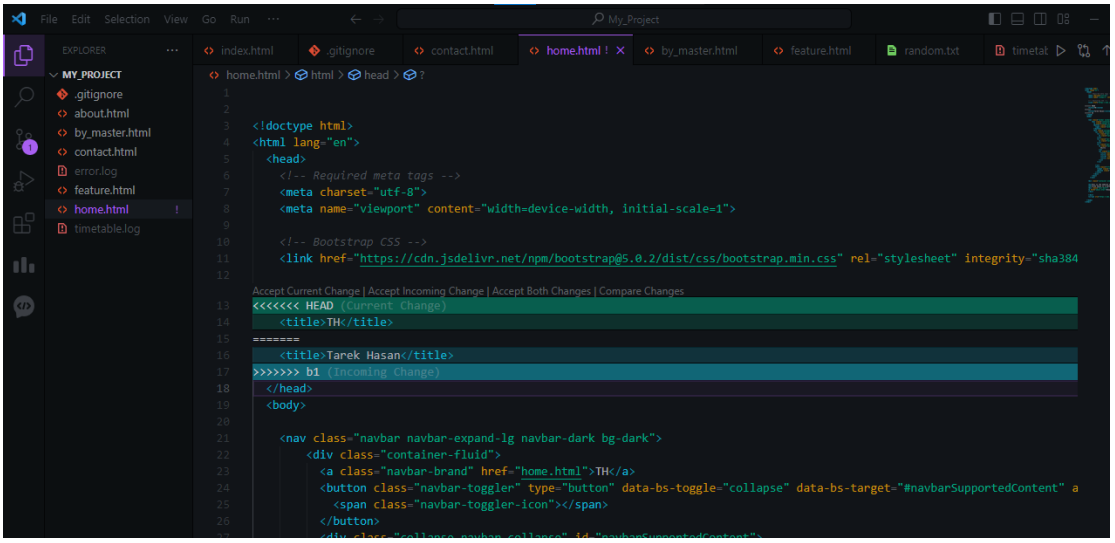
Let’s try to merge b1 into b2:



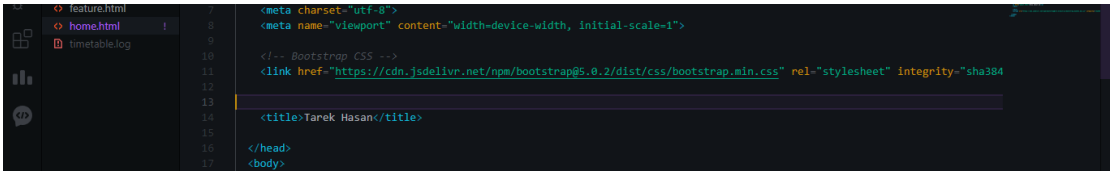
It shows the commit error.

▼ Resolving Conflicts

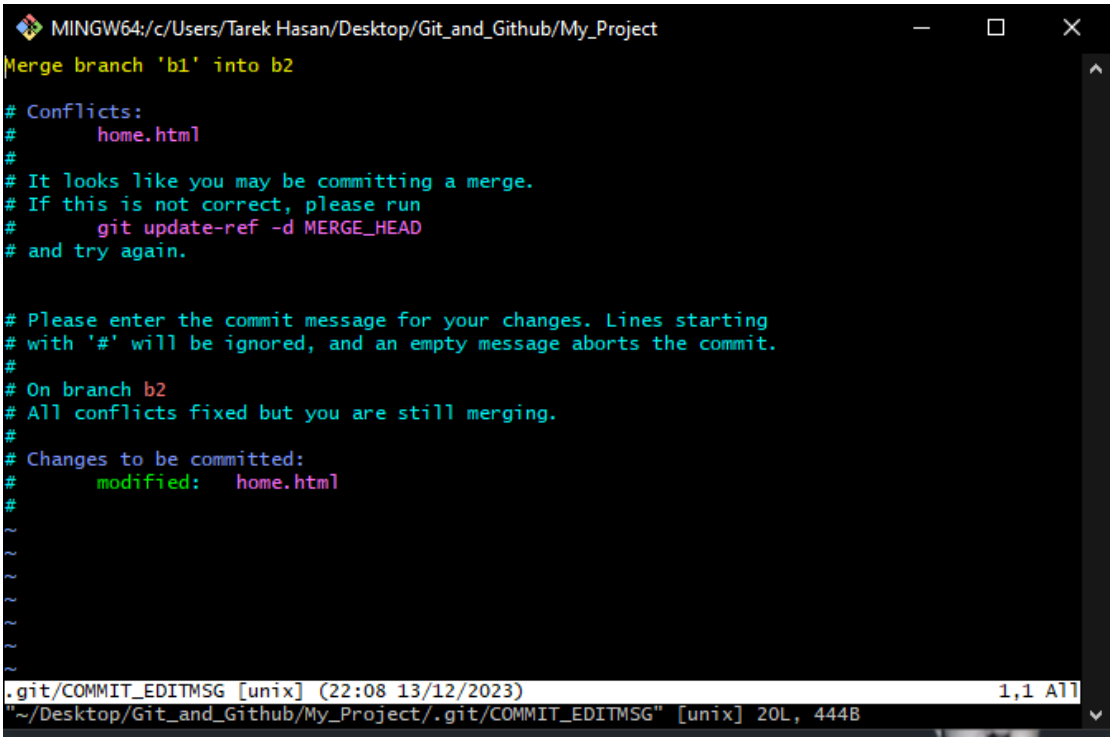
▶️ So we have conflicts in merging. Let’s look into the VS Code window:



It already shows in which things we are getting this conflict. It also shows the incoming data and current data. Also gives you the option to Accept current or incoming data. Let's say we want to keep the incoming data. But we want to do it manually to learn. Just delete the current data:

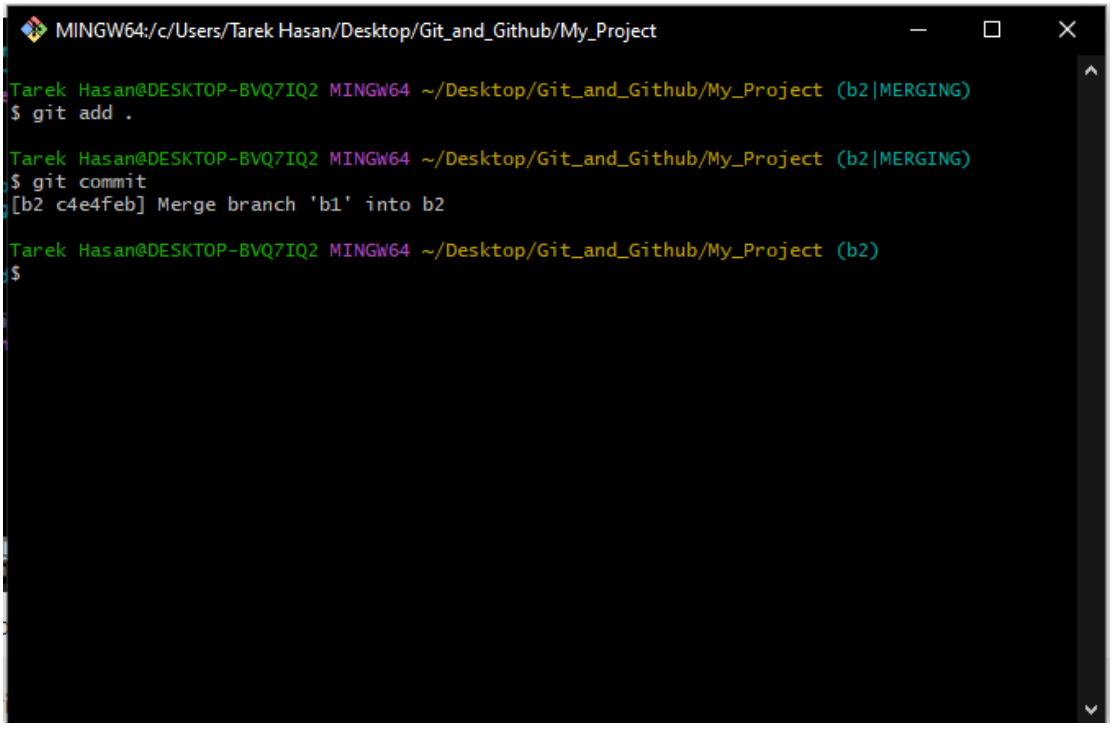


Like this. Now what we need to do is add this to the staging area and commit:



This window will show up. What we need to do is write **:wq** then press enter. This will make the commit.

> 📌 :wq means write and quite



Conflict has been fixed and merge has been done.

▼ **Branch Workflow**

▶️ There are 2 types of branches:

1. **Long Running Branches**
2. **Topic Branches**

Examples:

Long Running Branches:

> 📌 - Master
> - Development

Topic Branches:

📌 - Authentication
- UI

Discussion:

📌 **Long Running Branches:**
So the long running branches are those branch who stays till the project ends. You will have your master branch where you will keep the whole project. Then your development branch where you will develop the branch and test it then you will merge them into the master branch.

📌 **Topic Branch:**

Topic branches are temporary branch. Suppose you want to improve the UI of the homepage of your website. You will create a branch then improve the UI then you will merge it and then you will delete the UI branch. It is of no use after merging. So this is a topic branch.

▼ Rebasing

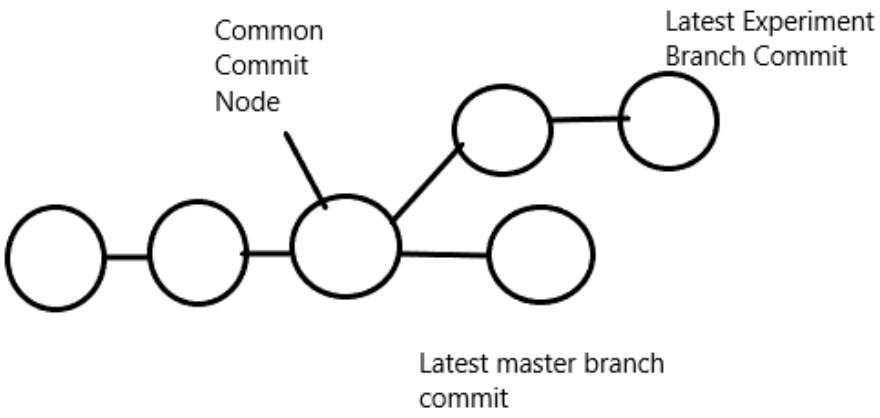
▼ Introduction to Rebasing

▶️ So what is rebasing actually? What is the difference between rebasing and merging? Well suppose you are in master branch. You have created a branch named experiment. You then did 1 commit in the master branch and switched to the experiment branch. Then you did 2 commits on the experiment branch. Now what if you want to merge experiment branch with master branch? You will switch to master then merge the branch right? What will it do? It will create a new commit in master branch with the changes of commits on experiment branch and the changes of commits with master branch. Now what will rebasing do? You don't need to switch to master branch while rebasing. Rebasing will basically create another commit in front of the latest commit of master and put all the commit in front of the commit that we made in master branch. The command is:

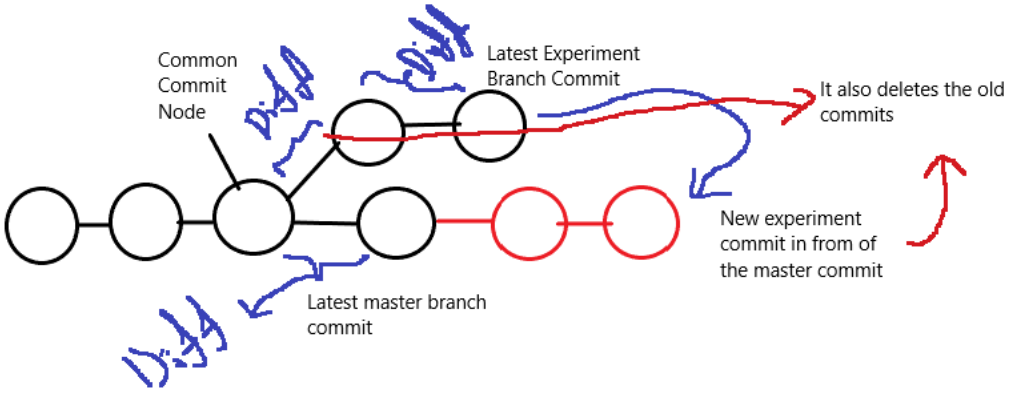
📌 git rebase master

▼ How Git Performs Rebase

▶️ Look at the picture below:

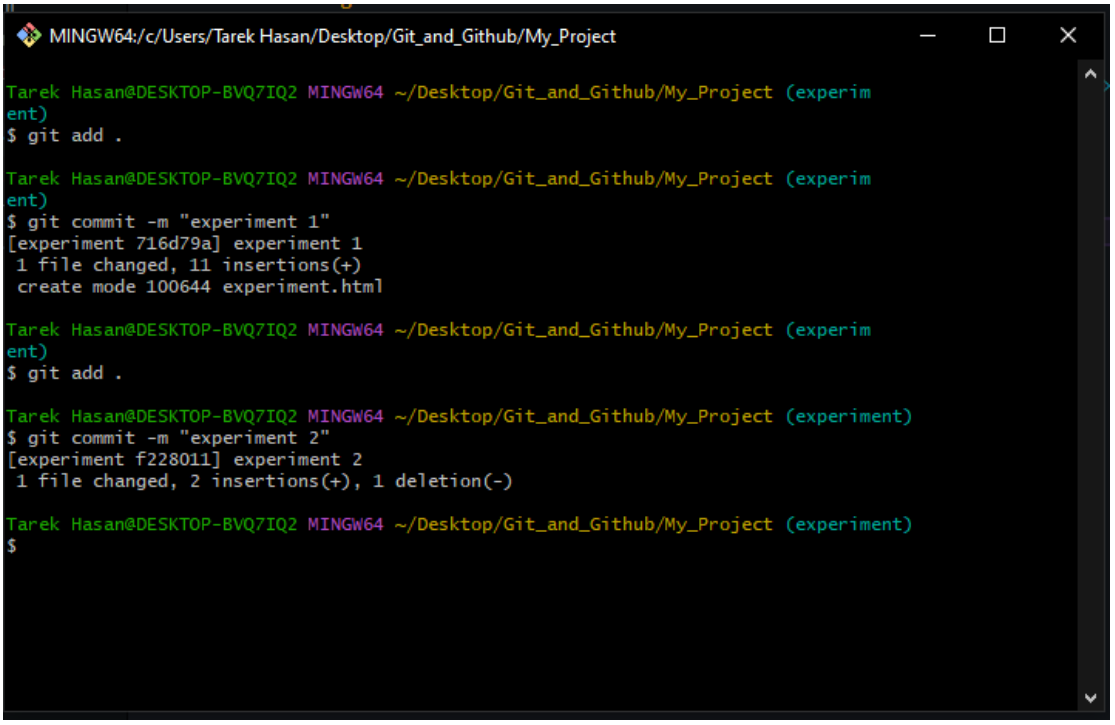


After rebasing:



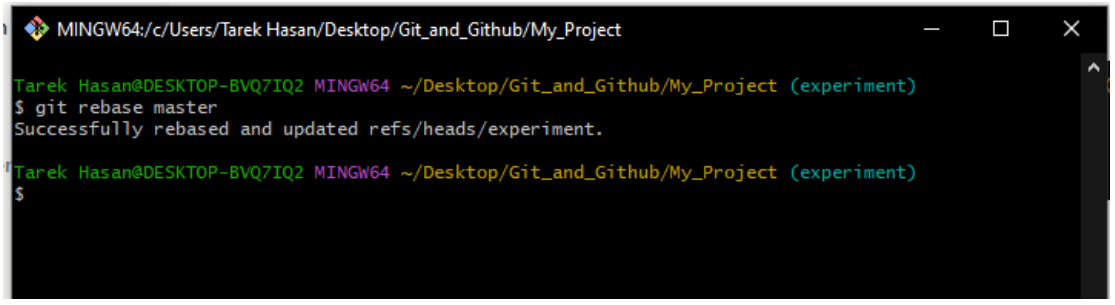
So what happens in rebasing? When we do rebasing it git checks the differences in the last commits of the current branch. Then checks if we have done some commits in the branch that we are rebasing in this case it's the master branch. As you can see we have 1 commit in the master branch so it checks the differences in that commit from the common commit node. Then is places the latest commits in front. But it's the experiment branches commit. The master branch commit stays on the same place but there is a way to bring it in the front. Let's look at the practical example.

▶️ At first we are going to create a branch named experiment and we will create a file named experiment.html and make 2 commits. Then we will switch to the master branch and make 1 commit. Then we will again switch to the experiment branch and rebase it. Let's do it:

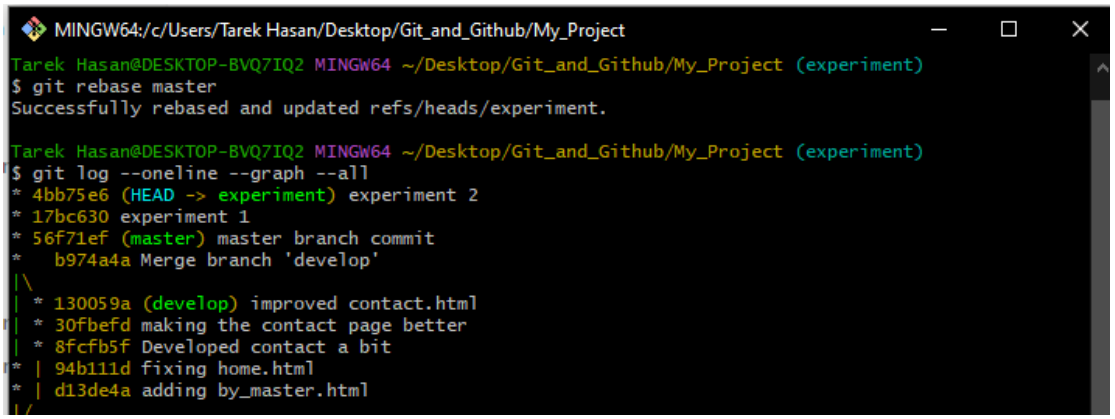


We finished the first part. Let’s do a commit on the master branch.

Now let’s switch to the experiment branch and rebase it:

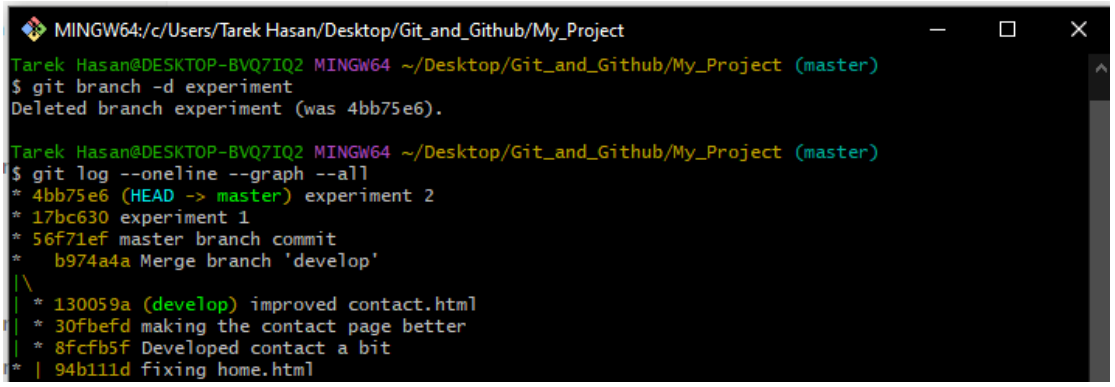


We did this and let’s checkout the in details git log:



As you can see the HEAD is one the experiment branch now. And also look carefully. What did we do? We first did the commit of experiment 1 and 2 right? Then we committed master branch. But look, it shows the commit of experiment1 and 2 in front. Not in the back. So that means experiment commit is in front of the master commit. Now how do we take the master commit in front of the experiment commit? To do that we need to switch to the master branch and merge the experiment branch. Let's see:

As you can see HEAD is on the master branch now. Let's delete the experiment branch and look at the log again:



▼ Key Points of Rebasing

▶️ What is the point of rebasing? Here are some:

1. No end difference between merging.
2. The log stays clean and clear.

▼ When you shouldn’t do rebasing

▶️ Suppose you’re working on a open source project where hundreds of peoples are contributing. You have your master branch and another branch on a commit of master branch named experiment. Now as it’s an open source project just suppose someone thought of decorating your experiment branch with new ideas so he created another branch from a commit of experiment branch. Now you have rebased the experiment branch with the master branch. Then the main problem begins. The person who created a branch from you experiment branch will have to marge it to the new commit of your master branch as rebasing deleted all the past commit of experiment branch and merged them into the master branch. He will have to find a similar branch of the main commit he created his branch from.

▼ Merge vs Rebase

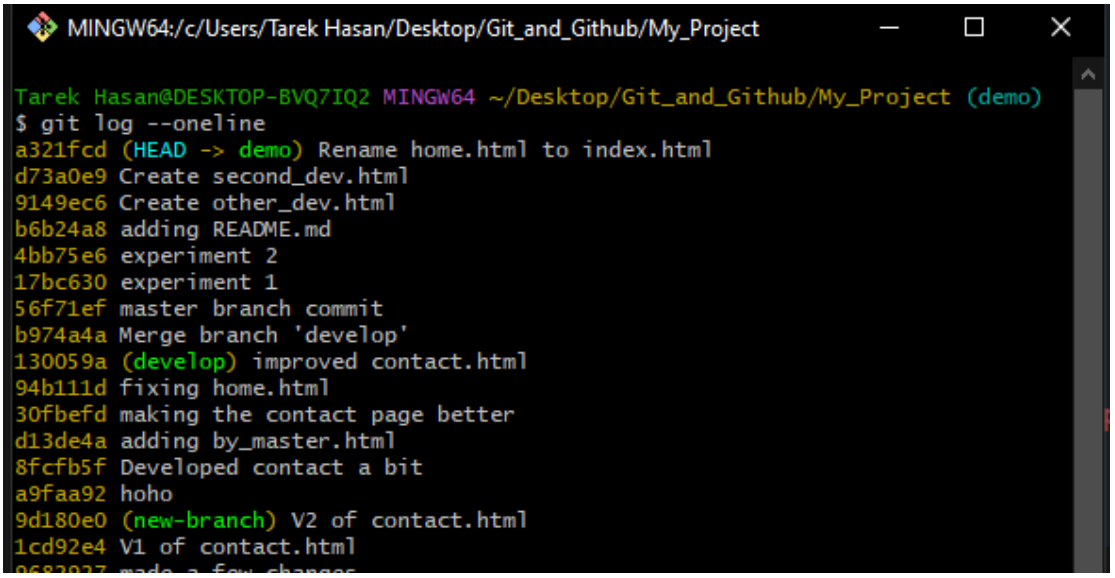
▶️ We should use merge when we want to be more transparent. We want to show people what we did in every step. Merge basically keeps track of everything we do from commit to everything.

▶️ If we want to hide some commits we can do that using the rebase method. As you have seen, we rebased something from the experiment branch to master branch. Then there was no existence of experiment branch. So yeah use it if you want to keep everything clean and non-transparent.

▼ Git Squash

▶️ What does exactly Git Squash means? Suppose you have 200 commits in your project. Now you don’t want these many commits which may seem like clutter to some. So you you to combine the last 100 commits to 1 single commit that will make your total commit into 101. Let’s see how can we do that.

Let’s look at a git log -



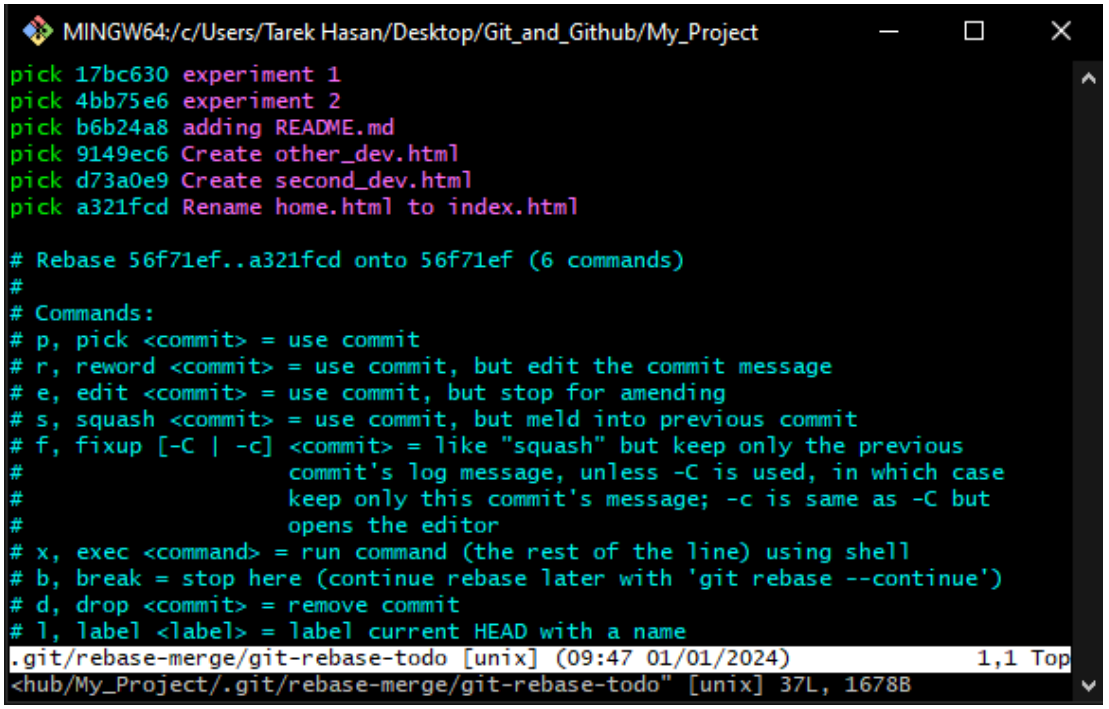
This is our git log. Suppose I want to combine all the commits till **[17bc630 experiment 1]** in one commit. That means all the commit after that will get merged into that commit. To do that we need to first run this command:

> 📌 git rebase -i HEAD~<THE NUMBER OF COMMITS YOU WANT TO SEE FROM YOUR CURRENT HEAD >

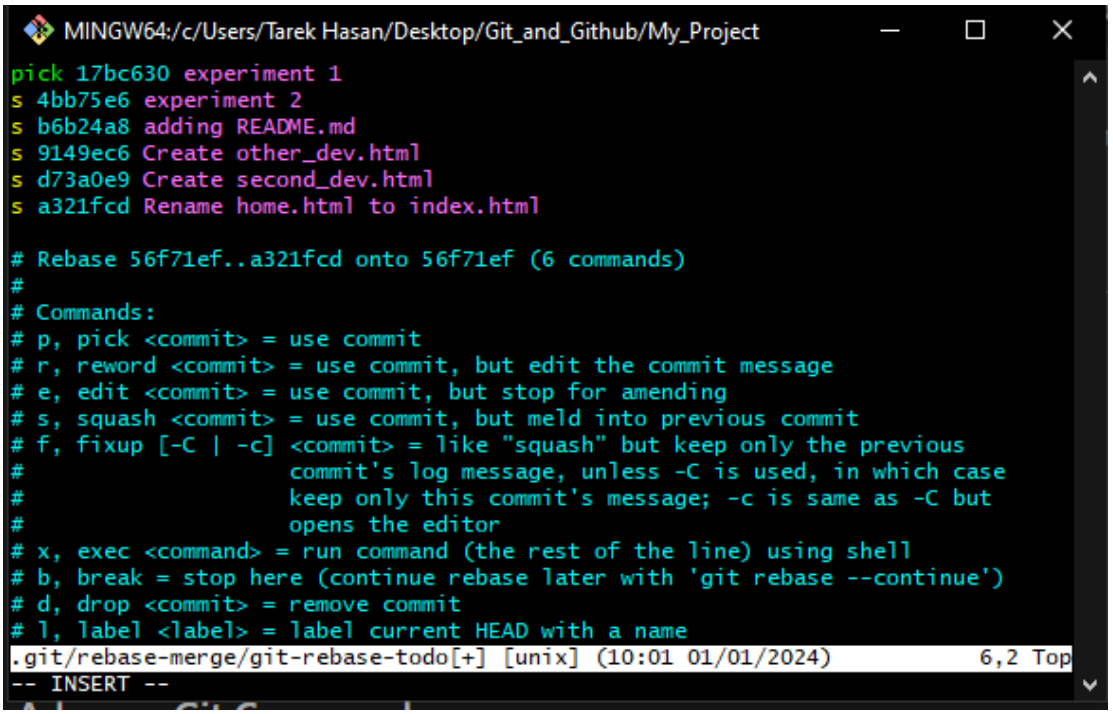
In our case from the HEAD position the commit with the message experiment 1 is 6 behind. So we will run this command:

> 📌 git rebase -i HEAD~6

Let’s see what happens:

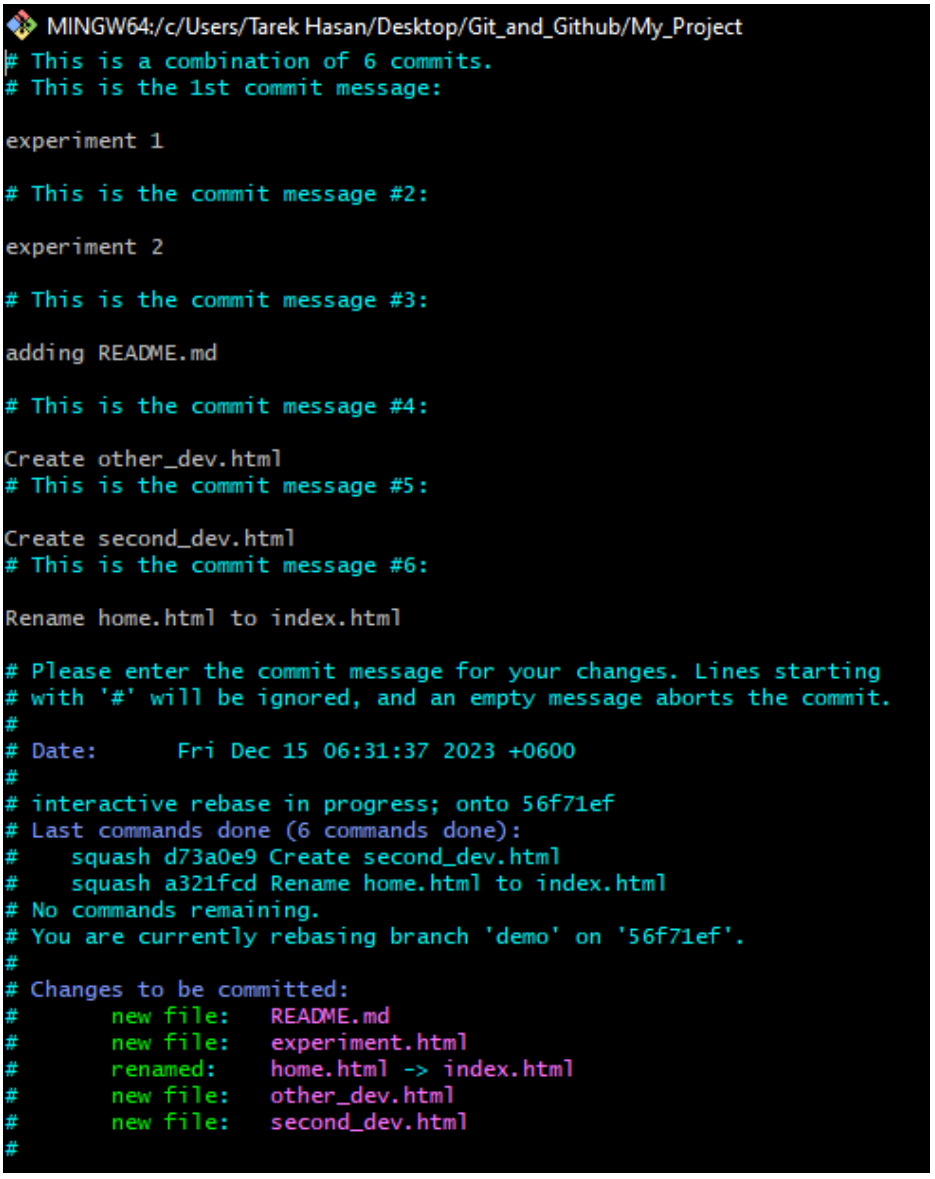


It shows all the commits till then. To do the squash we need to change the word pick to s. But which one should we command as s? We want to combine every commit after the experiment 1 commit into one right? So we should leave experiment 1 as it is and command s to all other:

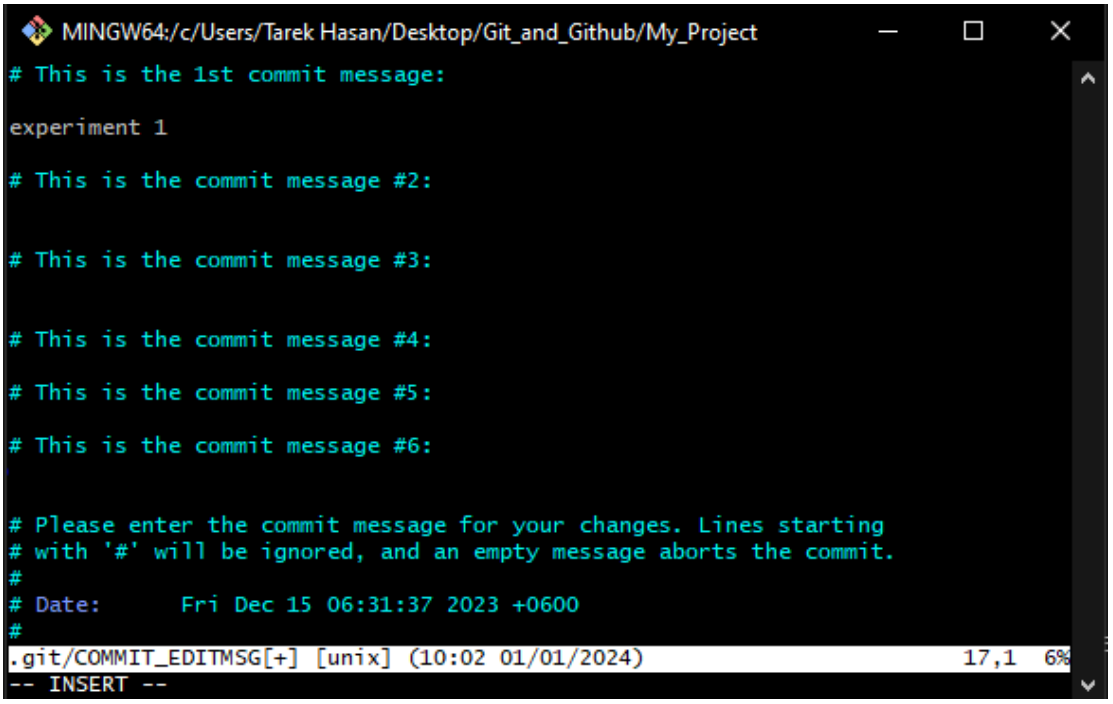


This is what we did. [To enter into edit mode press i and you can edit the file. To go back we need to press Esc key]

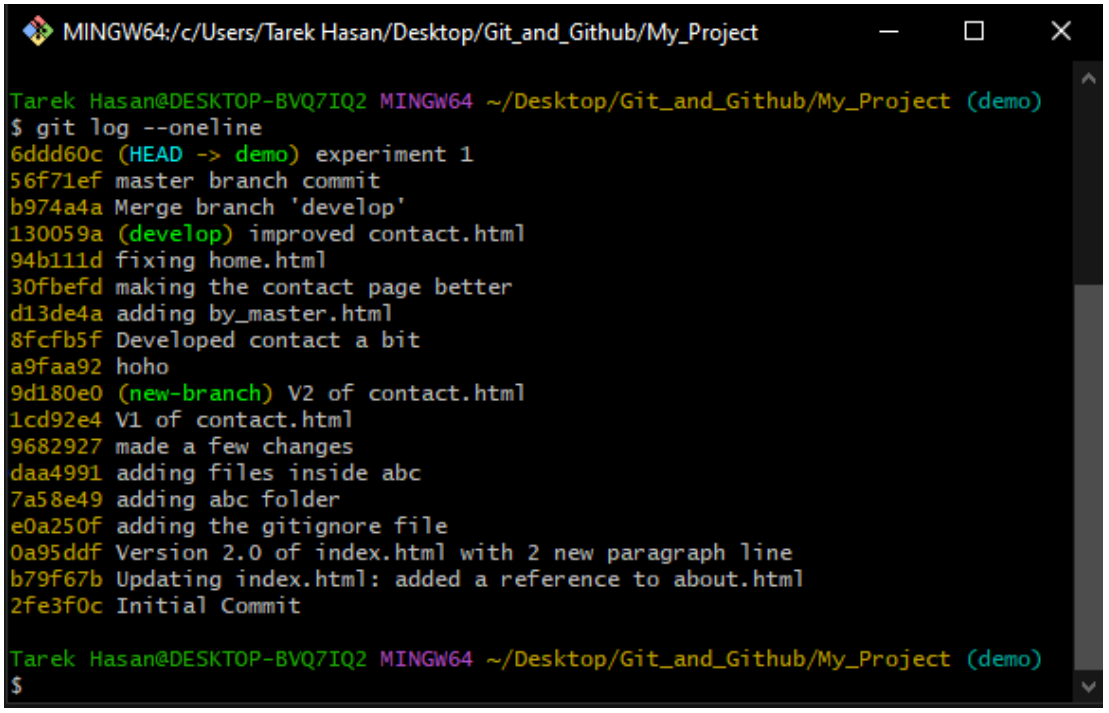
Now let's save the file. To save we can do :wq



As you can see it shows a message like this. As we want to keep it so experiment 2 we can remove all the other messages and save it as before:



Let's save it and look at the git log:



As you can see HEAD is now in the experiment 1 commit but we have all the changes in that commit made after that also.

▼ Introduction to GitHub

▼ Git vs GitHub

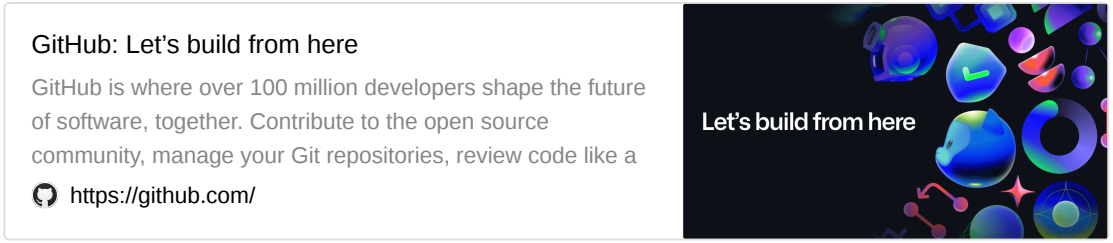
📌 Git is a version control system (VCL) that let's you keep track of your source code history and manage them.

📌 GitHub is a cloud based hosting service that let's you manage your git repository.

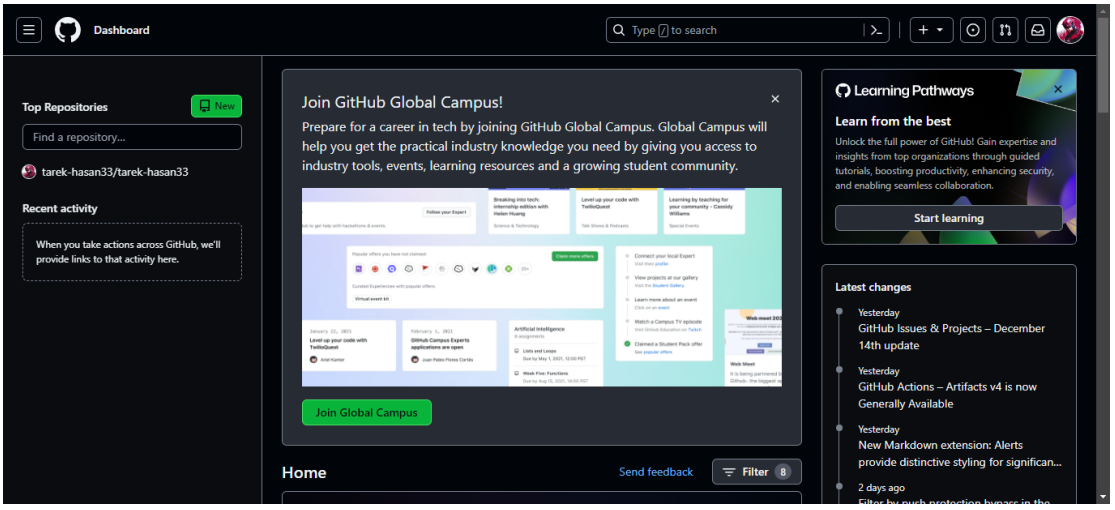
▶️ Like if you're working on a open source project or any project when so many peoples are contributing using git, then github is designed to manage them.

▼ Introduction to GitHub

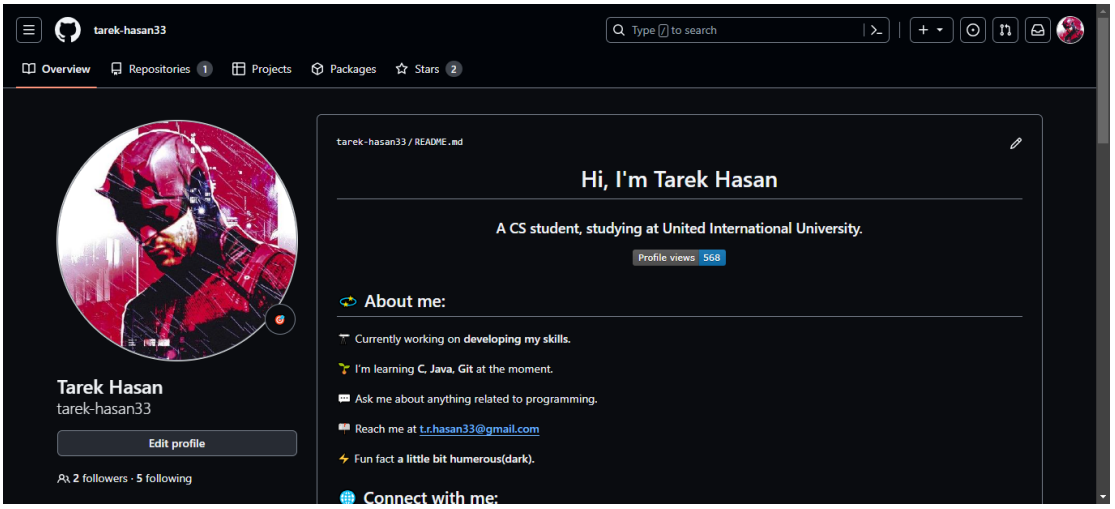
▶️ As we know, GitHub is a hosting website for git where we can host all our git repository. The address of the website is:



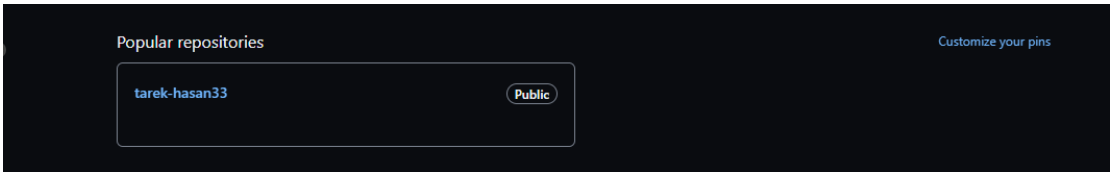
This is the dashboard or home page:



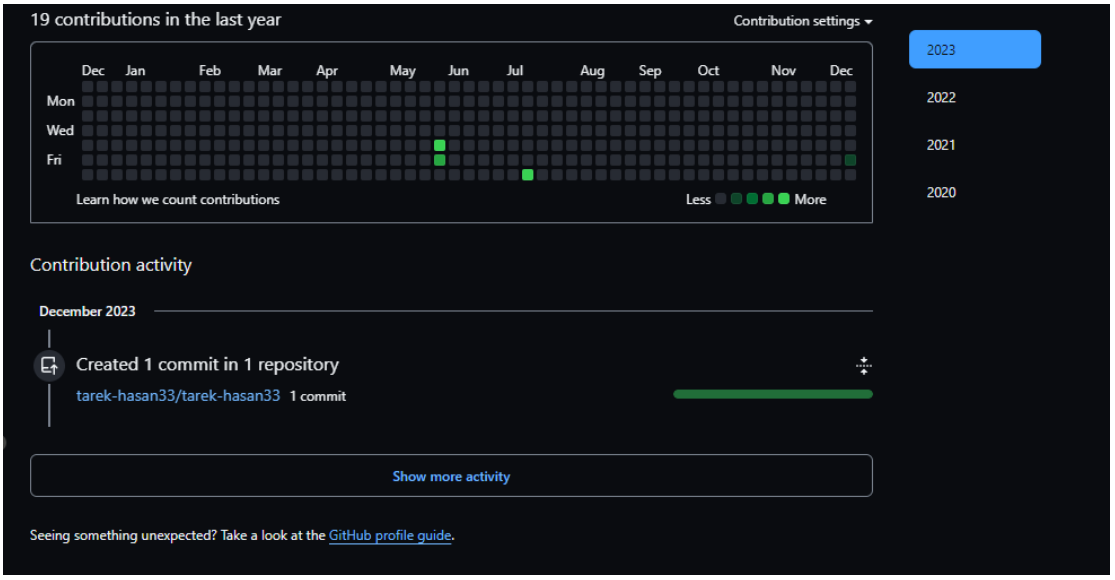
This will be your profile page:



Here you will find all you repository:

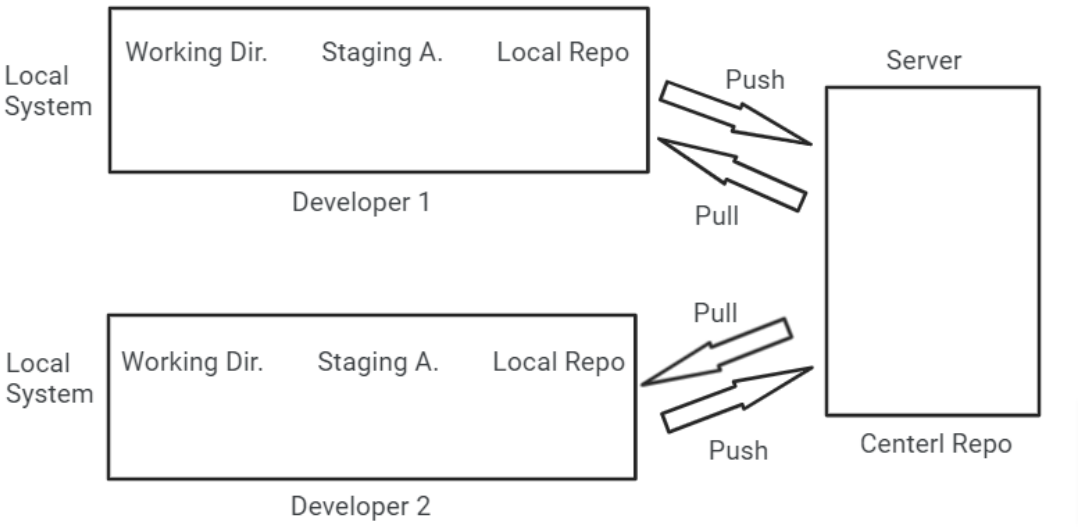


This is the contribution chart:



▼ Necessity of Central Repository

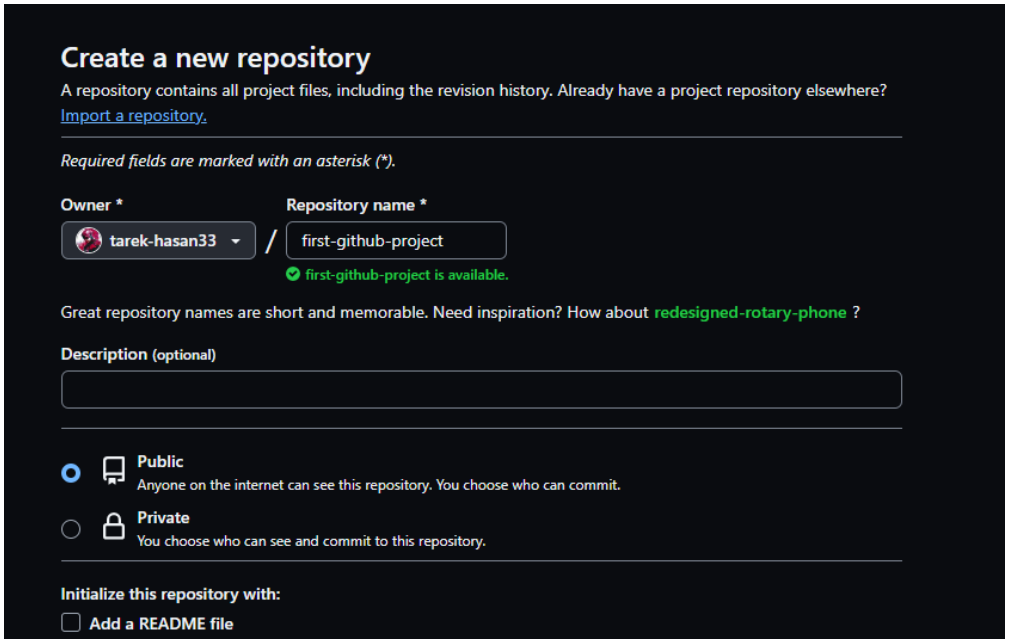
▶️ Look at the picture below:



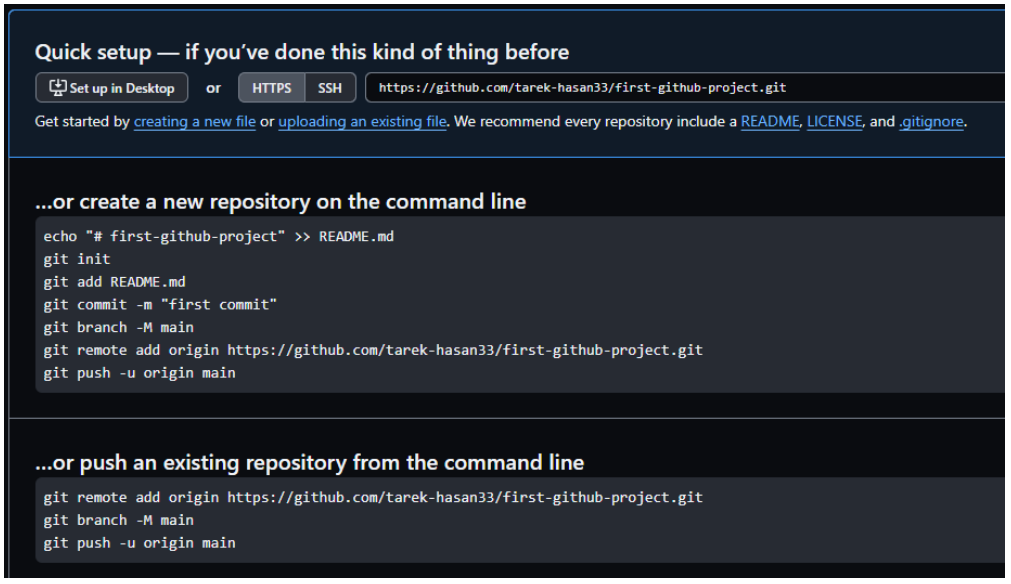
▶️ Why do we need the central repository? Suppose 100 developers are working on a project and everyone has the project file in their local system. But they want to develop a project together they need to contribute all together right? How can they do that if they only have the files in their local repository? How anyone else contribute in someone else's file? How will the project be developed? Here comes the importance of central repository. The main project stays on the central repository. Every developer can pull the files in their local device and work on them. When they finish working on the project they can push the file in the central repository. Every other developer will be able to look at the changes someone made.

▼ Creating a GitHub Repository

▶️ First we need to create a repository on GitHub which will be our central repository:



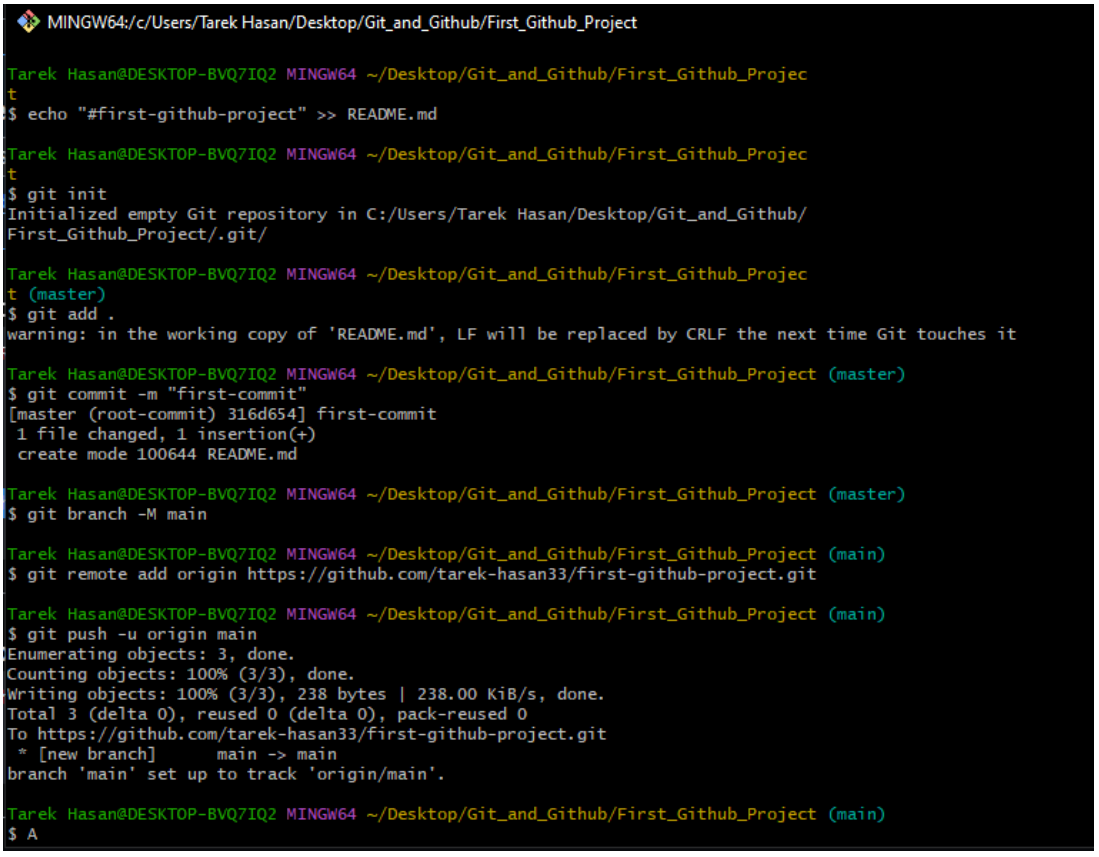
Then we will have a window like this:



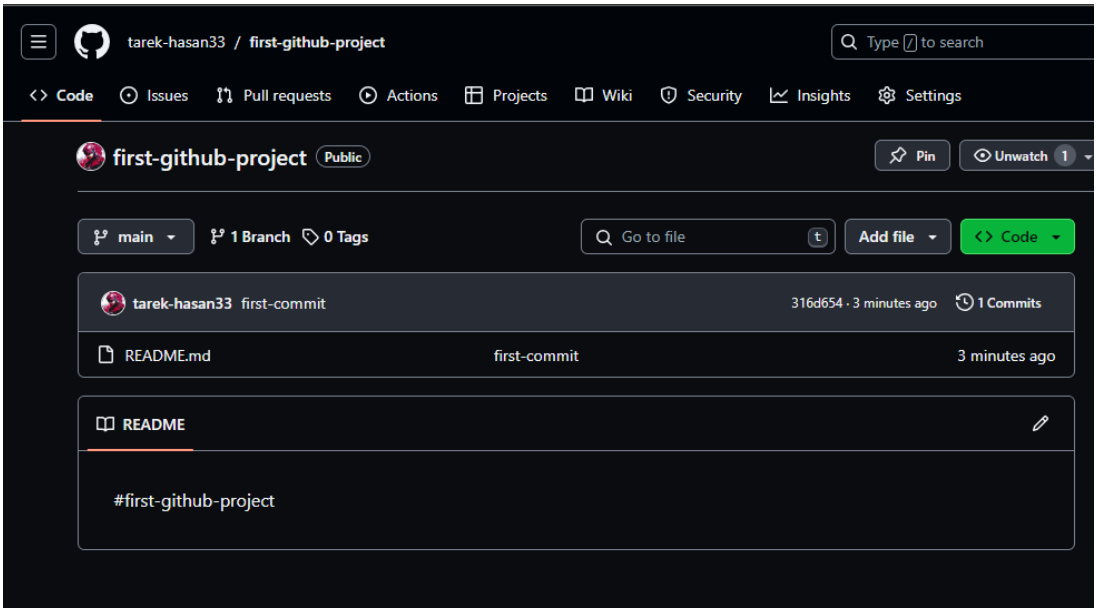
Then we will create a folder for our project in the local device. Now we need to run these commands one by one:

📌 echo "# first-github-project" >> README.md
git init
git add .
git commit -m "first commit"
git branch -M main [Changes the name of the master branch to main]
git remote add origin https://github.com/tarek-hasan33/first-github-project.git
git push -u origin main

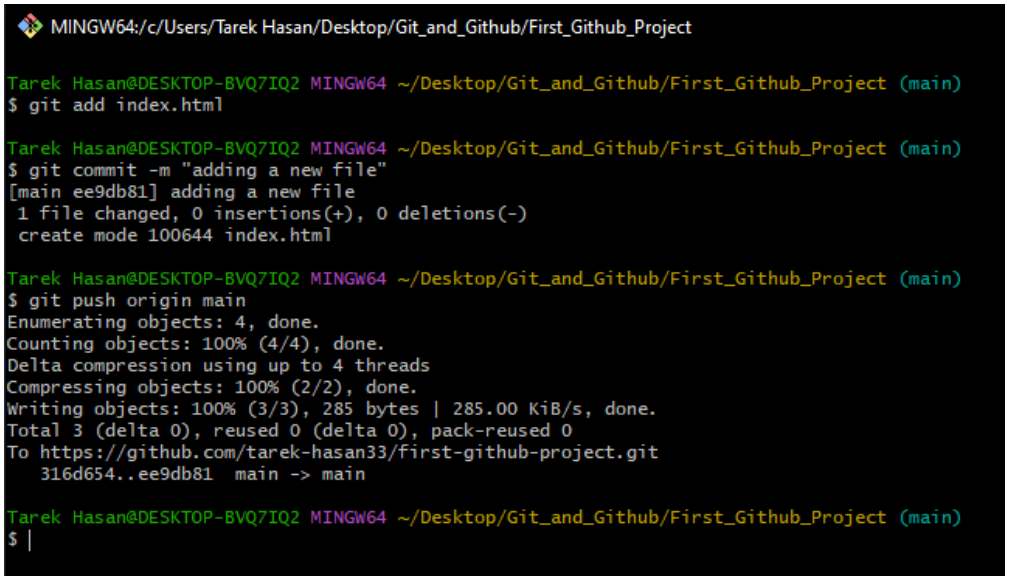
After running every command:



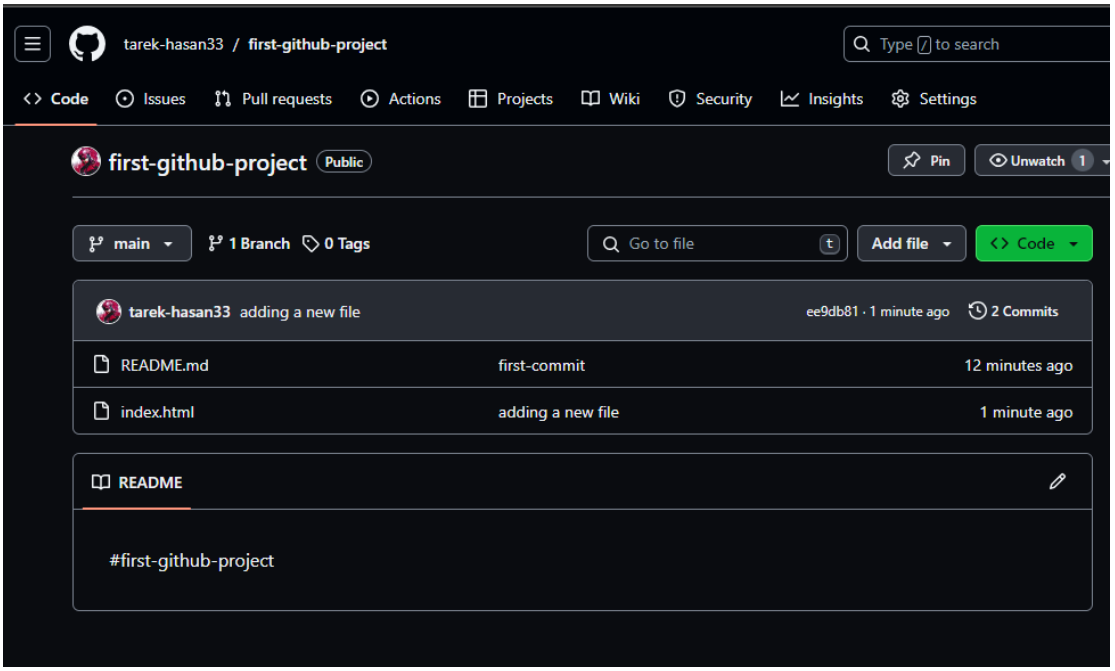
Here you can see in our new repository we have the file README.md:



Now let's test this with another file:



We have created a file named index.html then pushed the file. Now we will look into the github window:

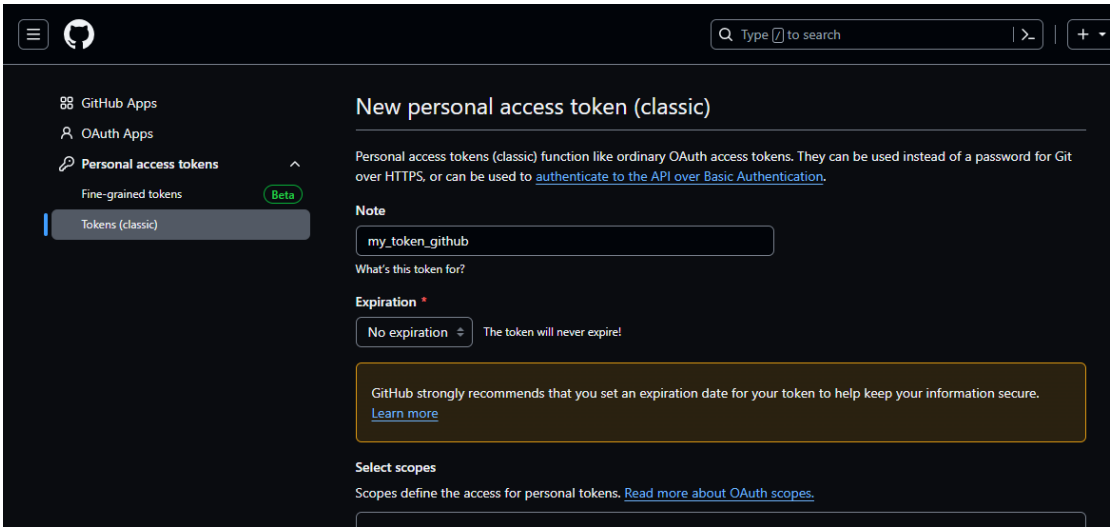


Now we have the file in github.

▼ Personal Access Token

▶️ When we try to push a code, github asks for authentication then you might have to put your password or give access from the phone app of github to push the code. That is why personal access token exist. You can create a token by going to:

> 📌 **Settings** >> **Developer Settings** >> **Personal Access Tokens**



Then you will get a code which you can use to authenticate. To get rid of the hassle of putting the password every time you push the code you can add this authentication key to the settings of your OS and it will do the work for you. To add PAT on your system please visit this link:

> Message "Support for password authentication was removed."
>
> I got this error on my console when I tried to use git pull:
>
> remote: Support for password authentication was removed on August
>
> https://stackoverflow.com/questions/68775869/message-support-for-password-authentication-was-removed

▼ SSH Key

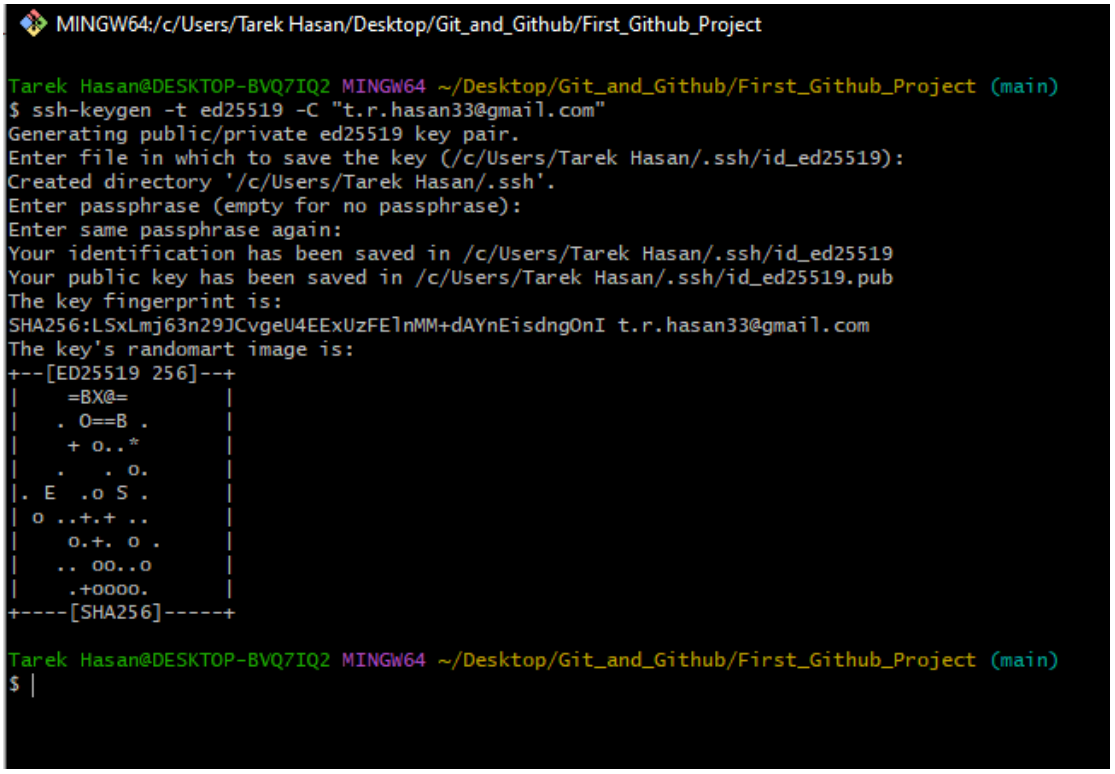
▶️ There is another way of authentication. It's called SSH key. We have to generate a SSH key and save it in the github profile then we can authenticate using the SSH key. It is more secure than PAT. To create a SSH Key we can use:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

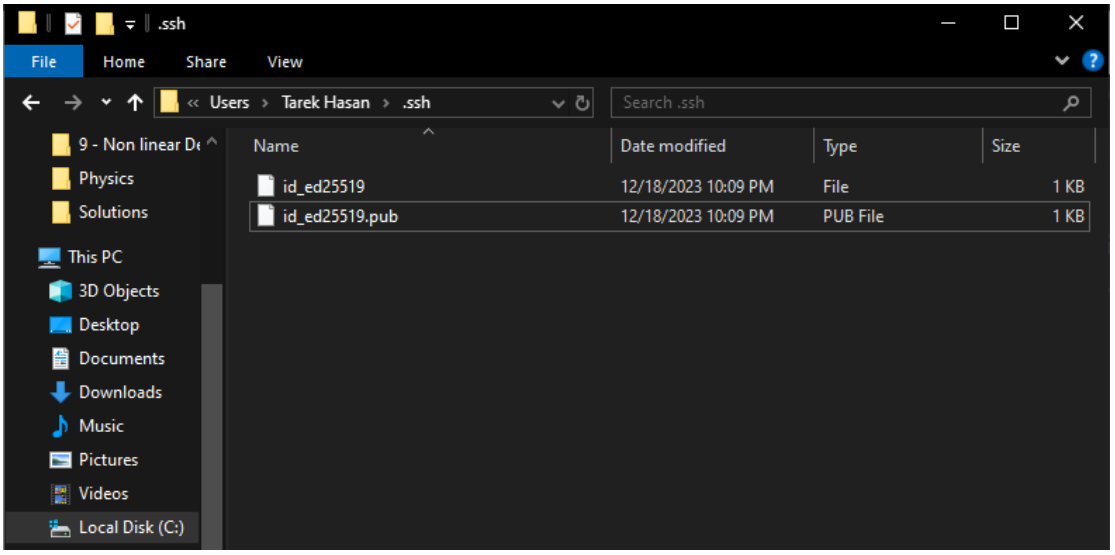
```
 Enter a file in which to save the key (/c/Users/YOU/.ssh/id_ALGORITHM):[Press enter]
```

```
> Enter passphrase (empty for no passphrase): [Type a
> Enter same passphrase again: [Type passphrase again
```

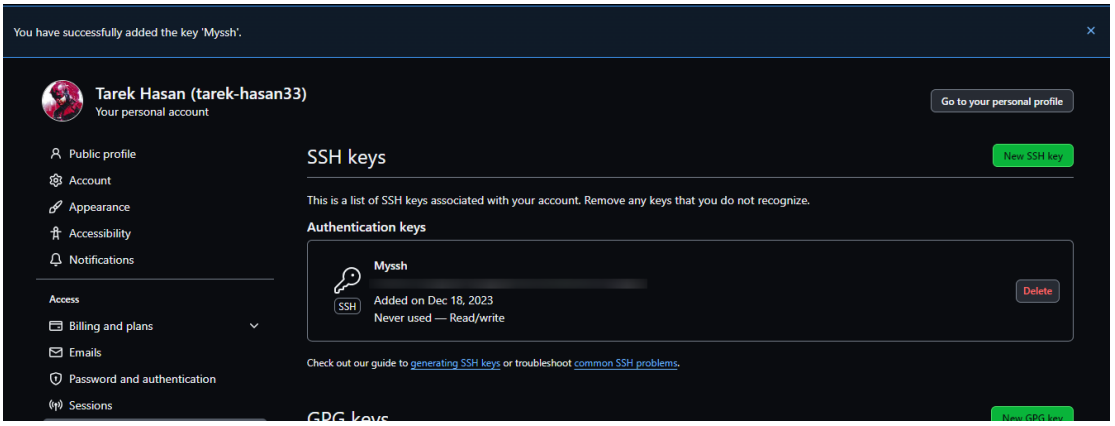
Let's generate a SSH key:



Now let’s look in to the file:



We have 2 files but 1 has pub on the end of it which means public. We need to copy everything inside the public file and set it on the github:



We can find the SSH option in setting page. As you can see I have already added the SSH key. Now to authenticate we need to run this command:

📌 ssh -T git@github.com

As you can see I have authenticated successfully. I had to put my passphrase too.

▼ Uploading an Existing Git Repository

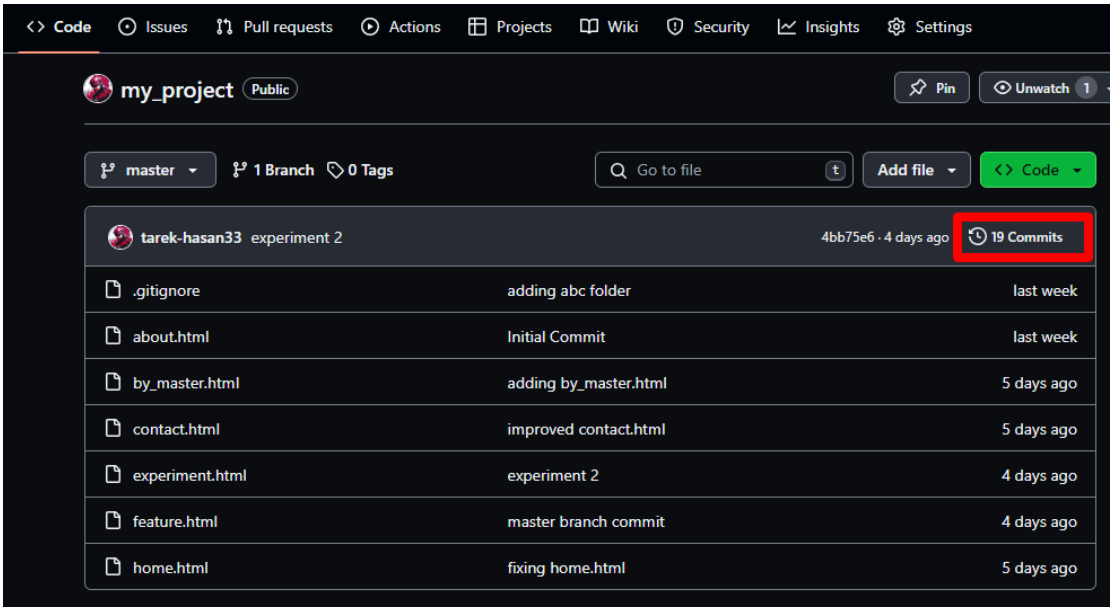
▶️ As we know we already have My_project as local repository. We will now upload it to Github. We need a few commands for this:

📌 git remote add origin https://github.com/tarek-hasan33/my_project.git
git branch -M main [we can also keep the name to master]
git push -u origin main

Let's run these commands:

```
MINGW64:/c/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git remote add origin https://github.com/tarek-hasan33/my_project.git

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git push -u origin master
Enumerating objects: 57, done.
Counting objects: 100% (57/57), done.
Delta compression using up to 4 threads
Compressing objects: 100% (52/52), done.
Writing objects: 100% (57/57), 7.51 KiB | 480.00 KiB/s, done.
Total 57 (delta 22), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (22/22), done.
To https://github.com/tarek-hasan33/my_project.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

Now let's check the github page:



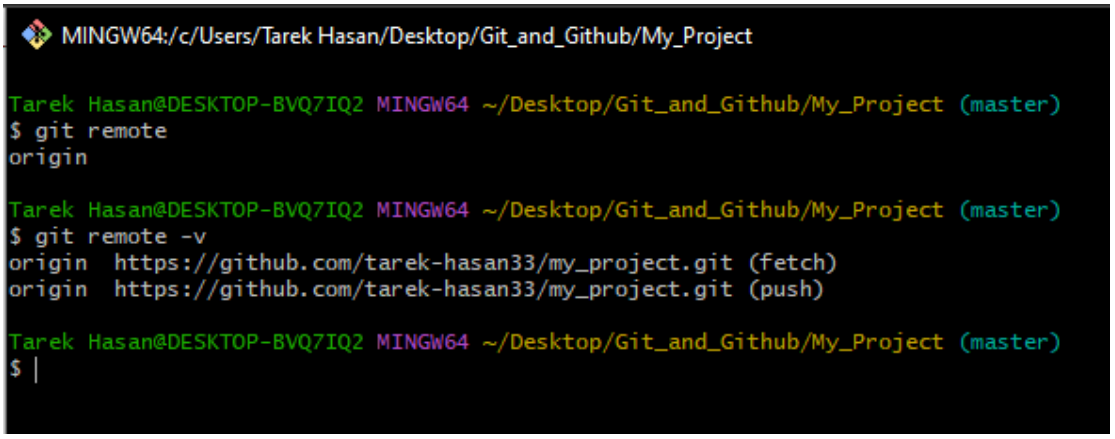
As you can see we have successfully uploaded the project. We also have 19 commits already which we made in the git.

▼ Remote

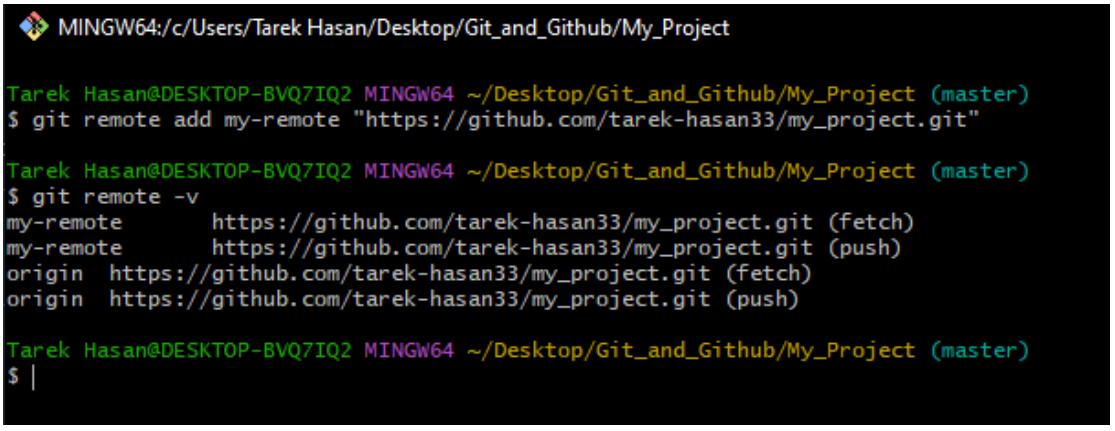
▶️ What is remote? Well remote is the thing that we use to pull and push the codes into github. You have already saw something like this:

> 📌 git remote add **origin <**LINK**>**

Here origin is the remote. Let's see our remotes in the project folder:



So **git remote -v** command shows the remote with the link. Let's add another remote:



We have added another remote. We can use my-project instead of origin now to pull and push.

▼ Cloning a GitHub Repository

▶️ So how can we clone a repository from GitHub? It's easy as it seems. There is a command for that:

> 📌 git clone "LINK OF THE PROJECT"

We can clone any github repository using this command.

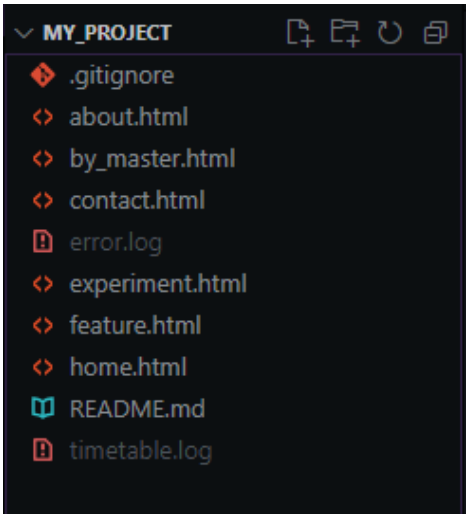
▼ Coding with GitHub

▼ Git Fetch

▶️ Suppose you are working on a project where 99 other developers are working too. You are done with your coding session today and went to sleep. You woke up the next day and saw other contributors added 10 more files to the project. Now

you need those files too on your local repository. We know that we can clone the central repo. But that's not the way. If we do that we need to again initialize everything from the beginning. That's why the **git fetch** command exist.

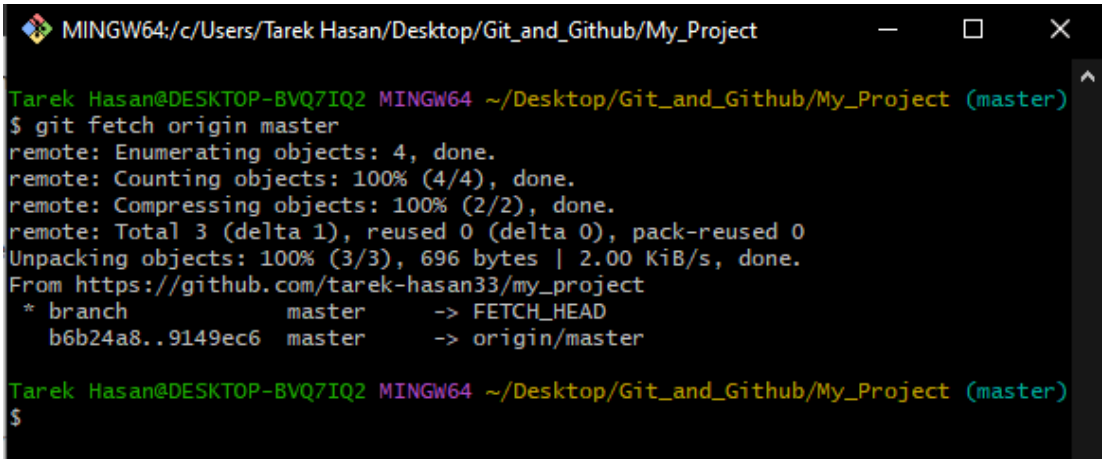
This is our my_project repo in the github page. As you can see we have added another html file named other_dev.



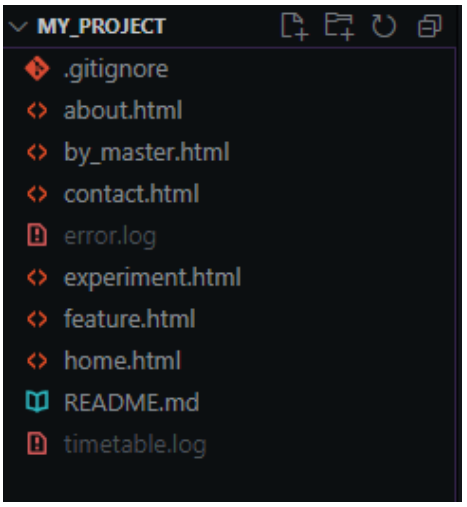
But we don't have that file in the local repository. Let's use the git fetch command and add it to the local repo. To do that we need to run this command:

> 📌 git fetch <REMOTE NAME> < BRANCH NAME>

In our case it is **git fetch origin master**



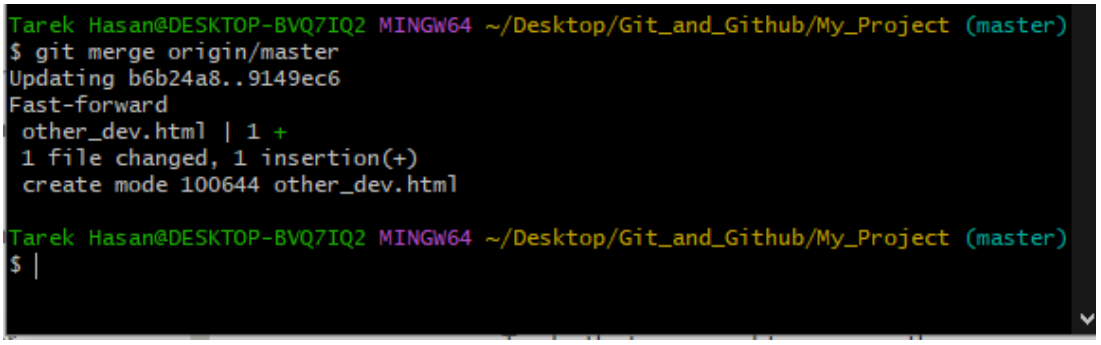
But file is still not in the working directory as you can see:



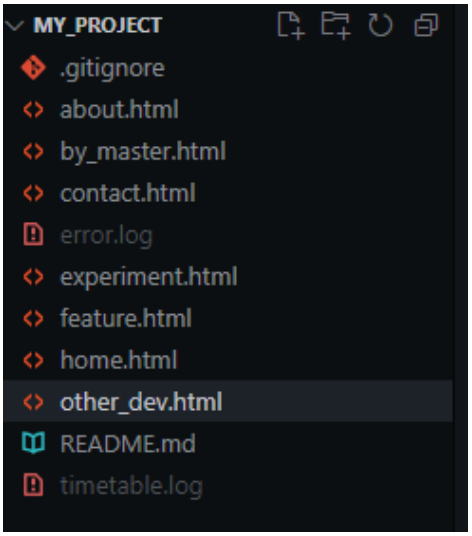
To do that we need to run another command which is the git merge command like this:

> 📌 git merge <REMOTE NAME>/<BRANCH NAME>

In our case the command will be git merge origin/master.



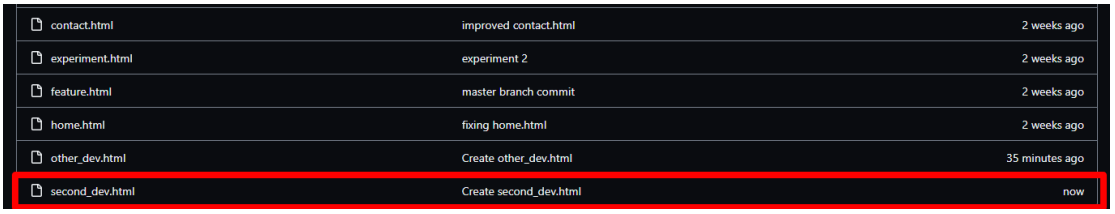
Let’s look at the working directory:



As you can see we have the file named other_dev.html.

▼ **Git Pull and [Git Fetch vs Git Pull]**

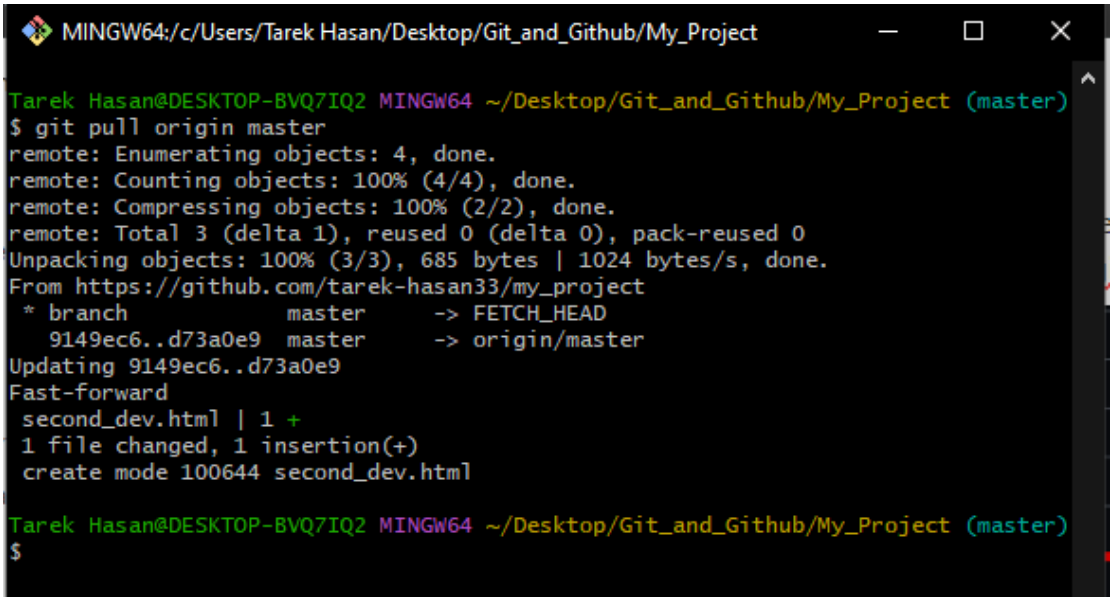
▶️ We already know what is git fetch. In git fetch we first needed to fetch the central repo then we needed to merge the files. But in git pull it does the fetching+merging at the same time. Let’s look at an example:



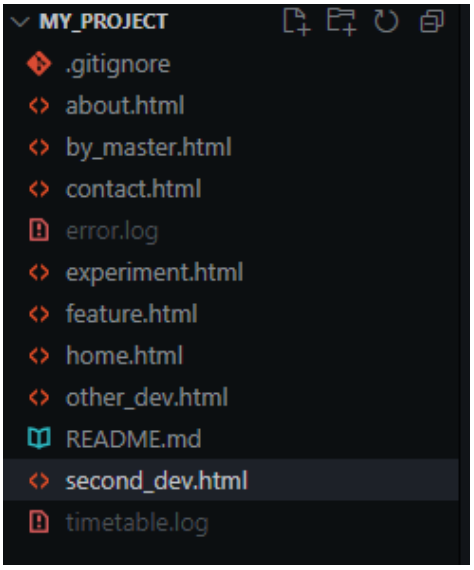
We have added a file named second_dev.html in the github. Now we want to pull it. We can run this command:

> 📌 git pull <REMOTE NAME> <BRANCH NAME>

In our case the command will be **git pull origin master**



Let’s look at the working directory:

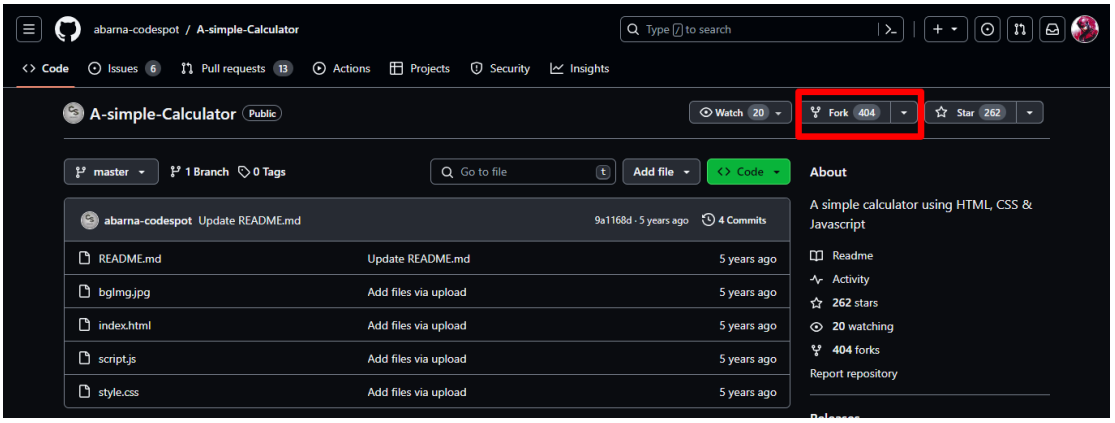


As you can see we have the file.

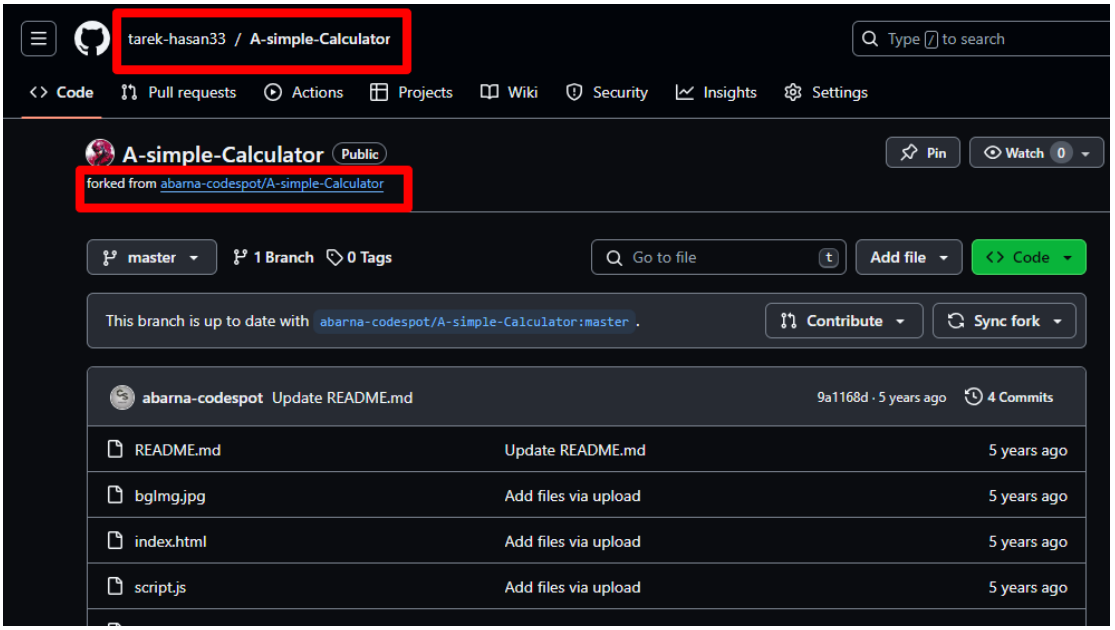
> ⭐ **But which is better? Git fetch or Git pull?**
>
> These are 2 different thing to do the same thing. But if you want something fast you can use git pull as it does 2 work with one command and faster. But if you want to be on the safe side without any merge conflict you can use Git fetch as it doesn’t add the file to working repository without explicitly merging the repo.

▼ **Fork**

▶️ What does fork means? Suppose you want to work on a project which already exist. I mean you want to work on that project on your own. Let's say you want to work on https://github.com/abarna-codespot/A-simple-Calculator project. You can fork it into your own github profile. Then you can clone it and work on it. Let's look at the practical example:



Let's fork this into our own profile.

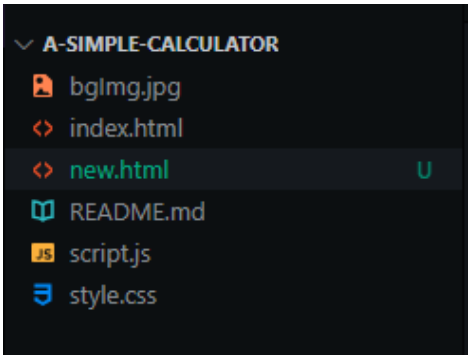


Now this is in our own repository. We can now clone this central repo to our local computer and work on it or modify it as we like. So that's what fork means.

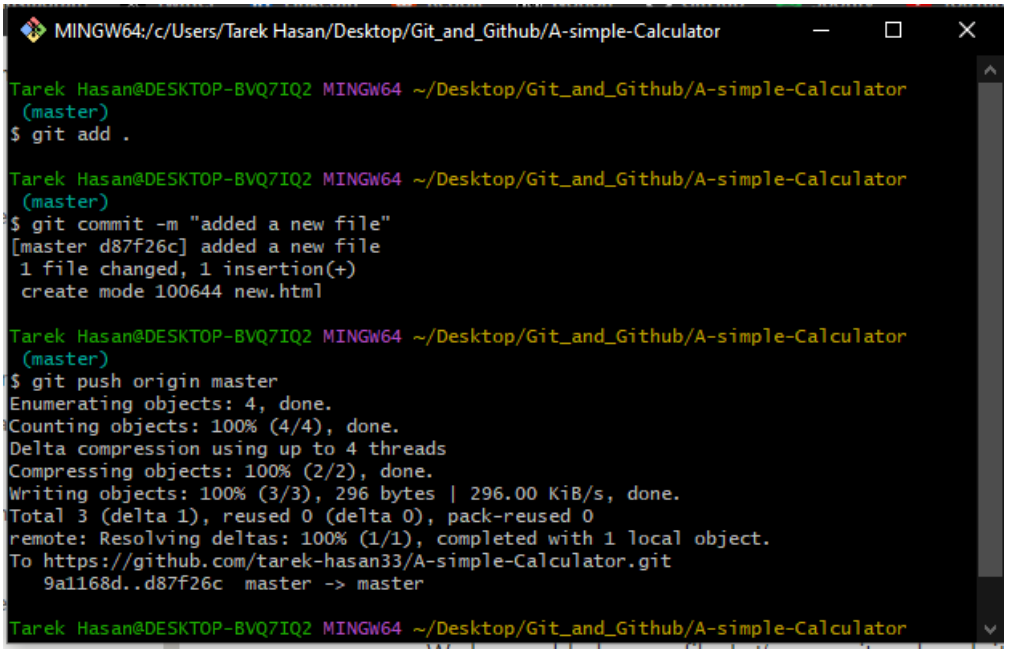
▼ Pull Request

▶️ Till now we have known about forking. Now let's see what pull request is. We already forked the https://github.com/tarek-hasan33/A-simple-Calculator repo in out account. Why did we fork it? Cause we want to work on this project. Let's clone the forked repo into our local machine.

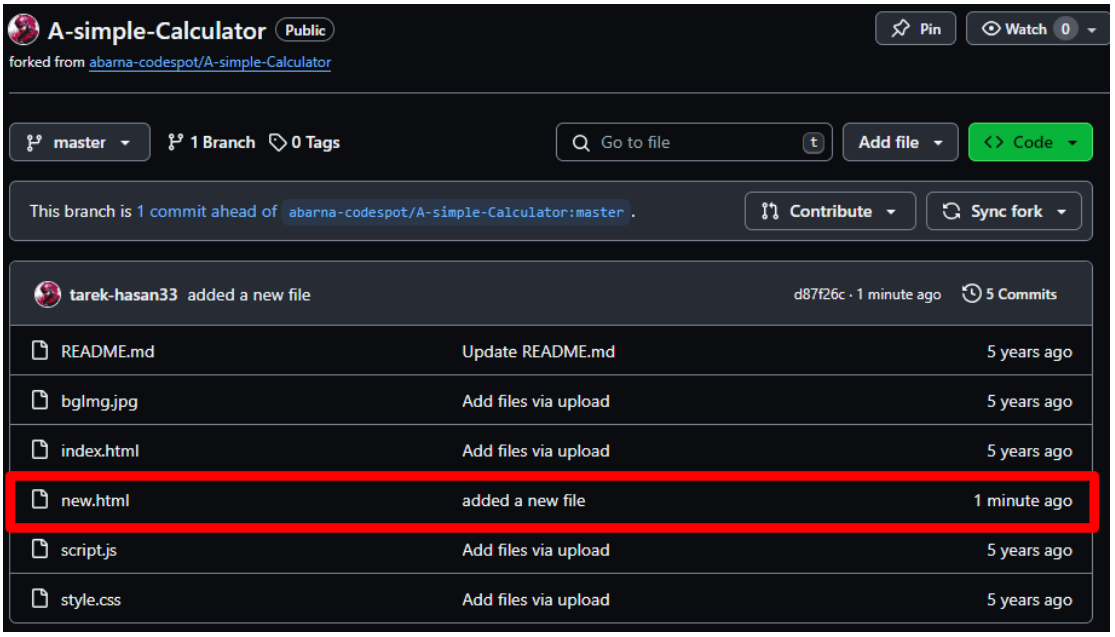
Let's open the project in VS Code and and add a file name new.html.



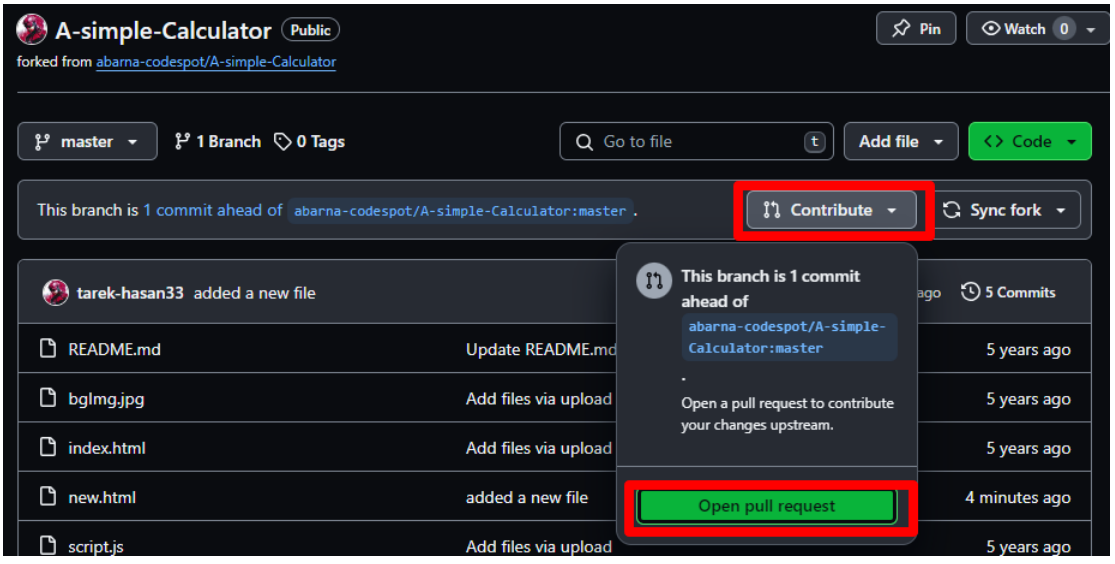
We have added a new file. Let's commit and push it to the forked central repo:



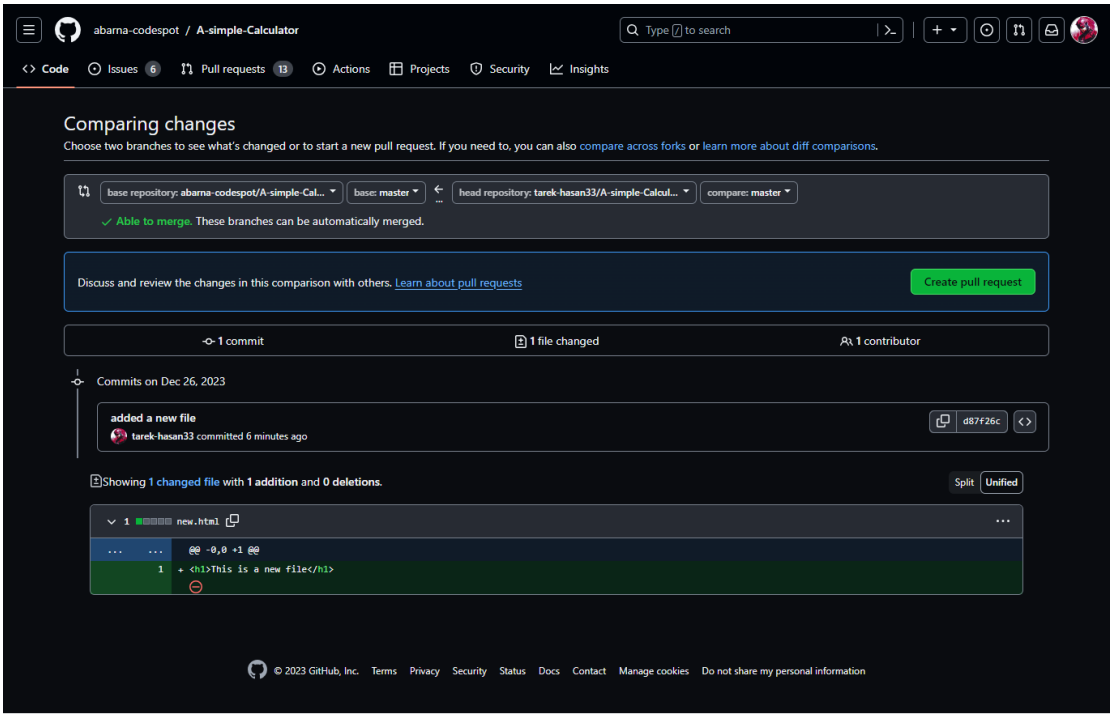
Now let's look at the GitHub page:



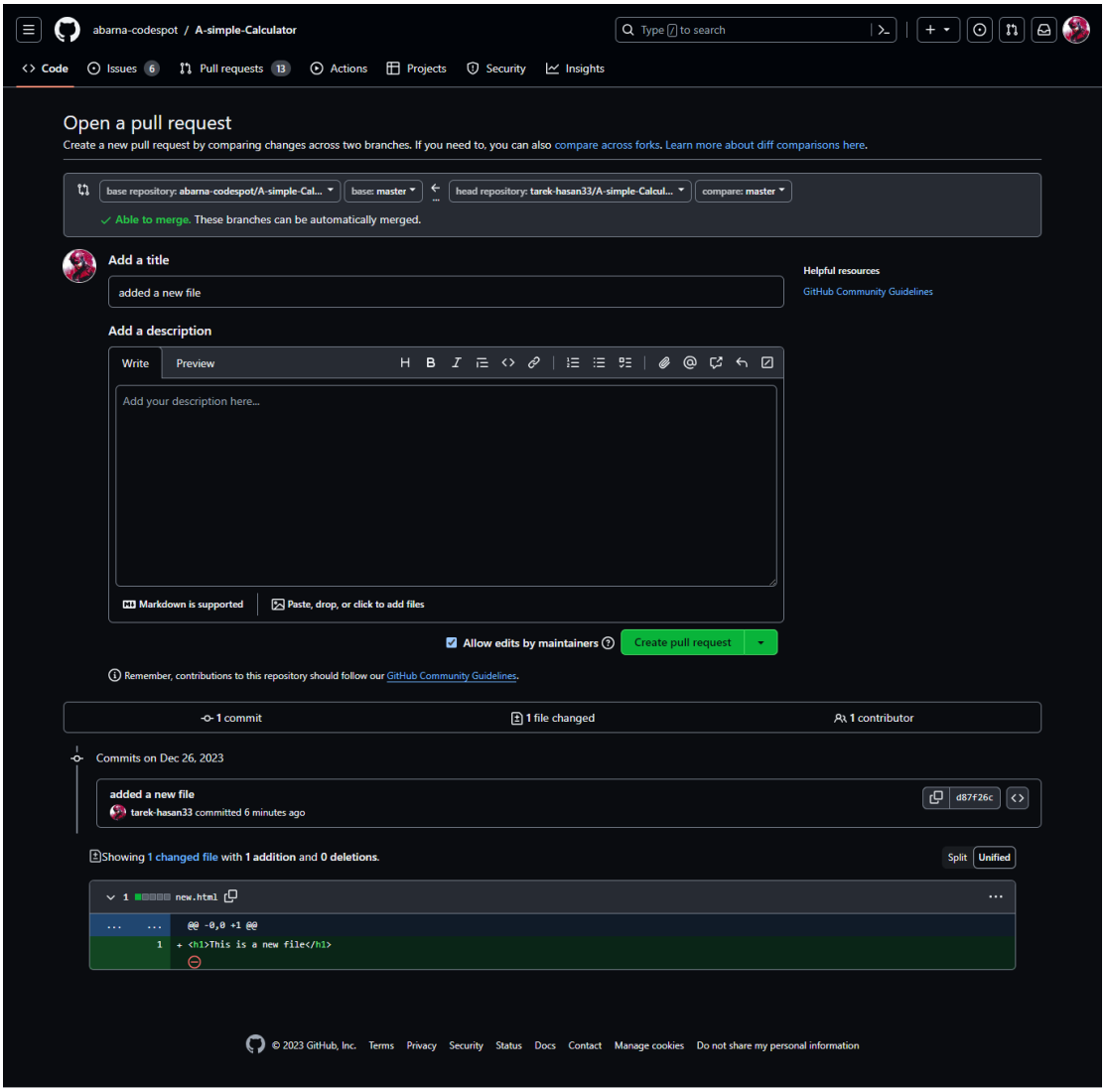
As you can see we have the new file. But the thing is it is the forked repository which is in my account. We haven't contributed yet. What do we need to contribute to the original project? I mean how can we add the new feature that we created to the main project? Yes you guessed it right, by requesting the author to pull the repo and merge it to the original project. We can do the pull request like this:



It will take us to a page like this:



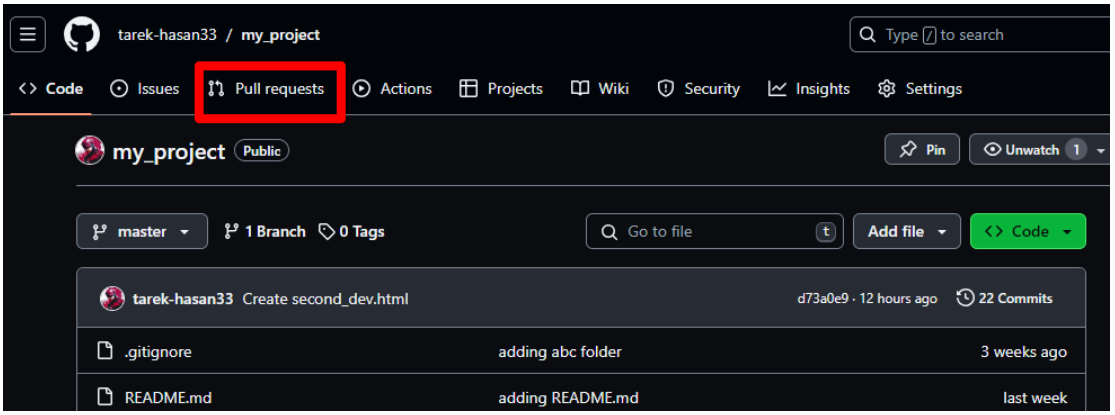
Then we can click the create pull request button and it will take us to this page:



We can now add the title and give a through description of what we are going to add and what have we changed. Then we can create a pull request. If the author thinks that our feature is addable then he will merge the changed to his repo.

▼ Merging a Pull Request

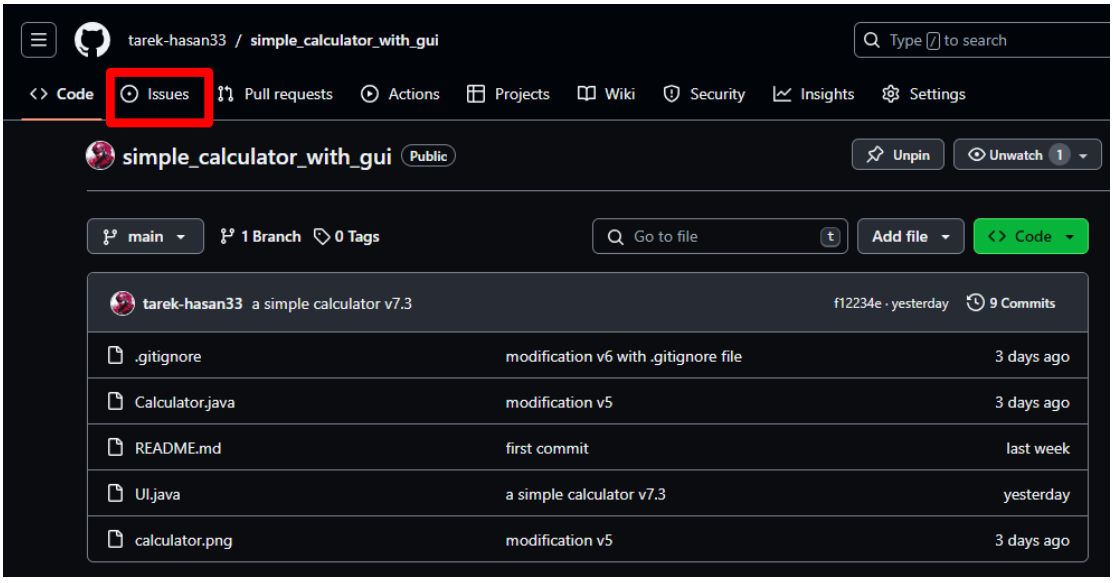
▶️ This is only required if you're the owner of a central repo and someone wants to contribute to you project. If someone sends you a pull request it will show up like this:



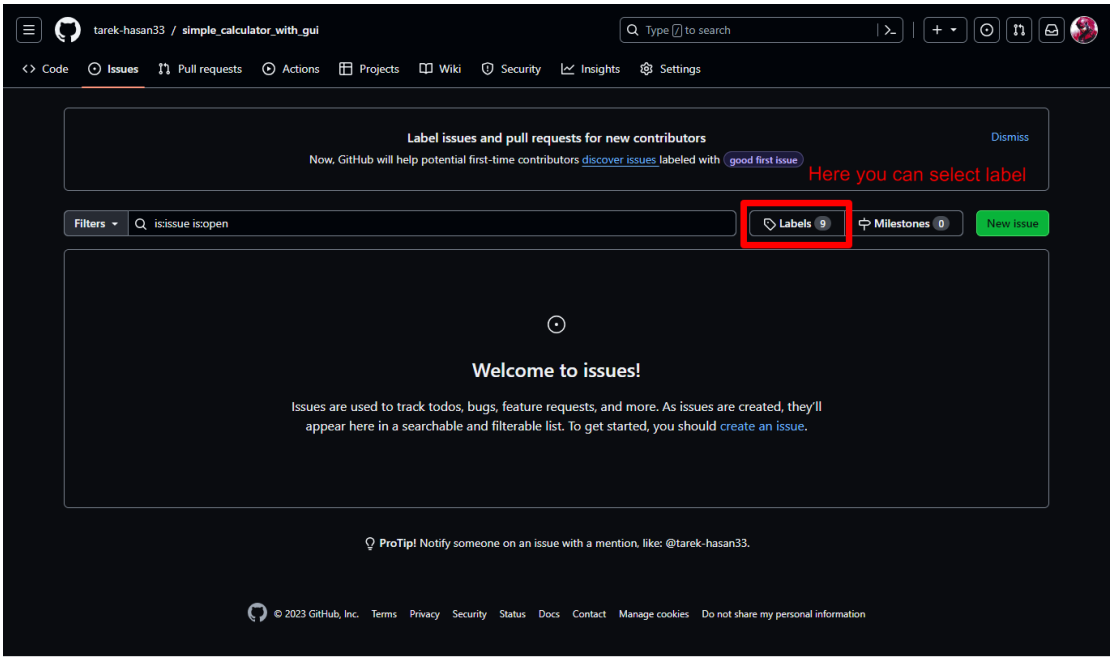
After going to there you'll have the merge option. We are not owner of any repo or no one is contributing that's why we can't show the full demonstration.

▼ GitHub Issues

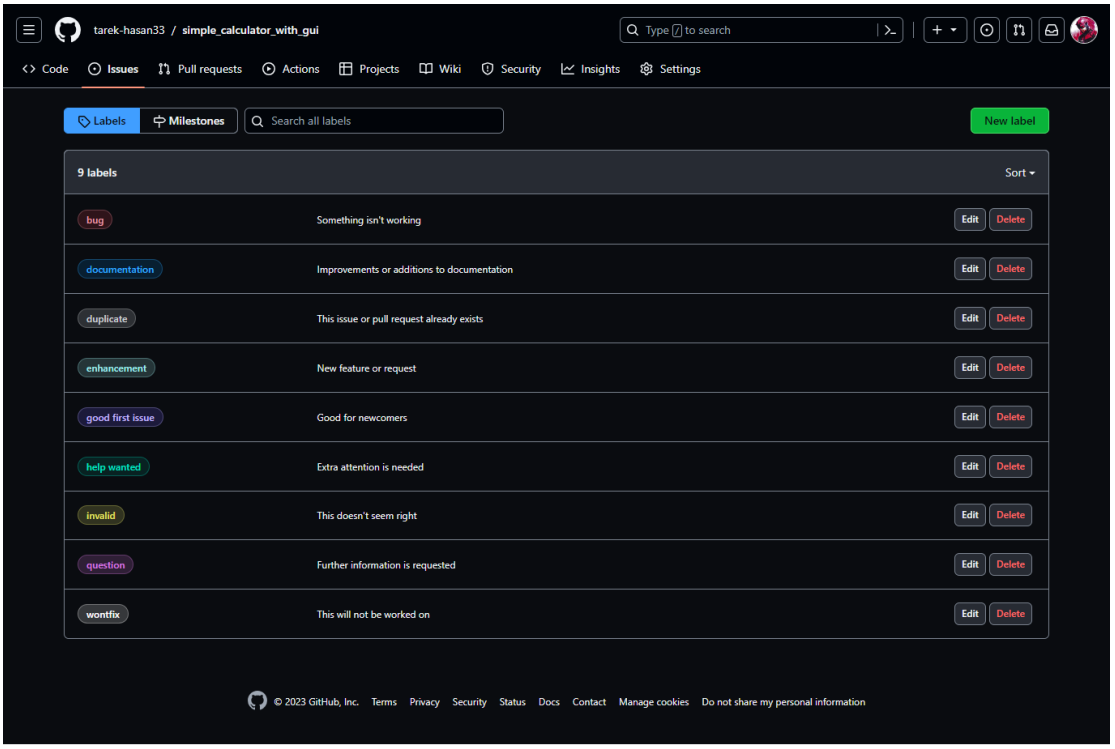
▶️ So what is GitHub issues? It is a functionality to create a issue for you project so that other developers may want to contribute. Let's look at an real life example:



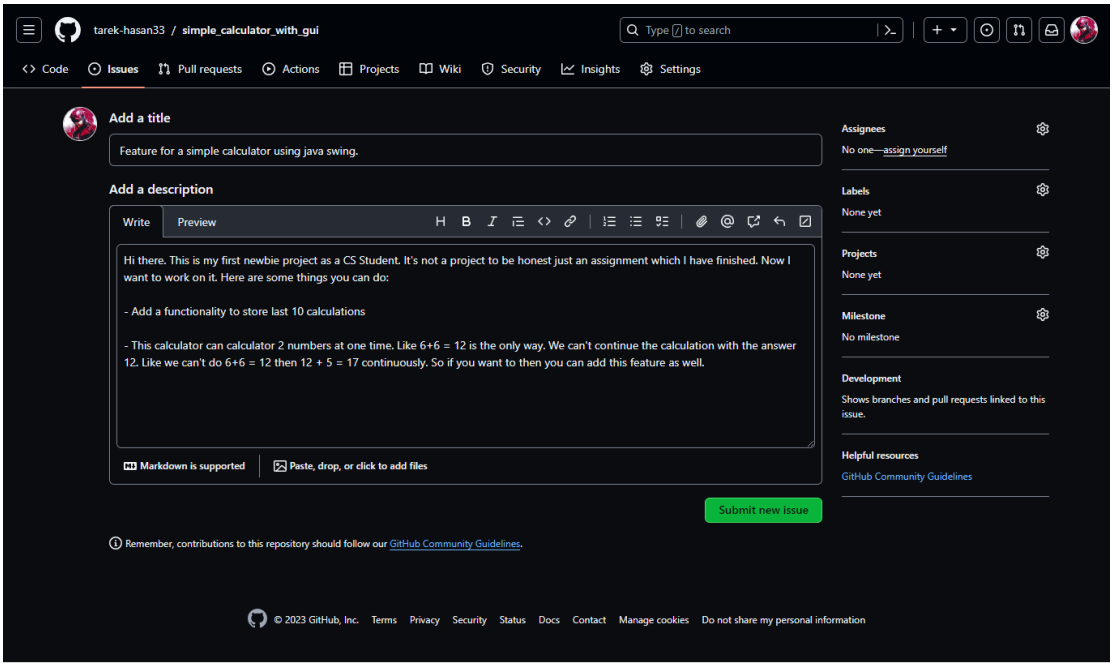
It is a project of mine and here you can see the Issues button. Let's see what is inside of it:



Let’s explore the label option:



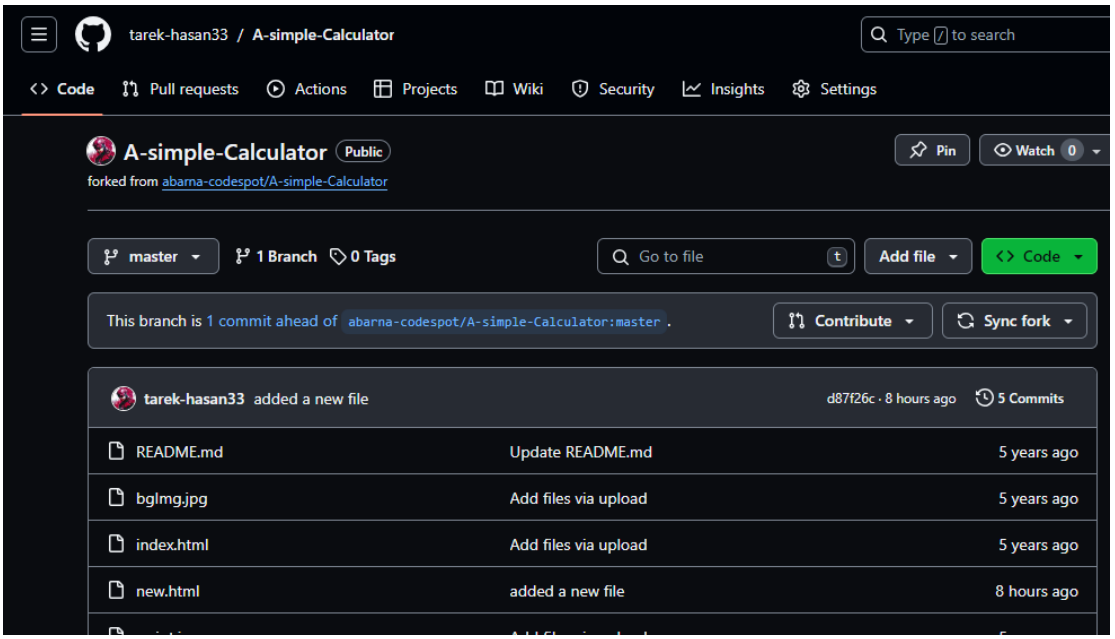
Here you can select different types of labels for you issue. Let’s create an issue for this project. After clicking the create issue button a page like this will appear:



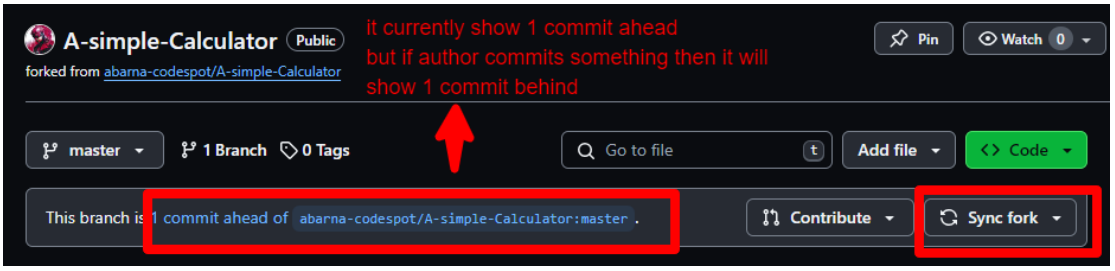
I have filled the sections and requirements for my need as you can see. Now if someone wants to add a feature then he/she can fork the repo and work on that after that they can create a pull request.

▼ Syncing Git and GitHub Repository with Upstream

▶️ So what does this mean? As we know we have the below repo as a fork:

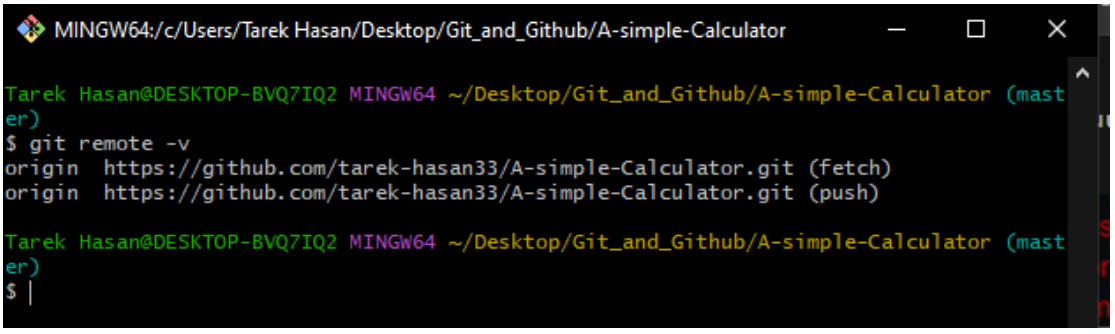


Now we are working with the project. Suppose the main author added a new file to the main repo. Remember what we have is a fork of that main repo. So whatever the author does with the main repo it will not change anything in the forked repo. So how can we always be updated?

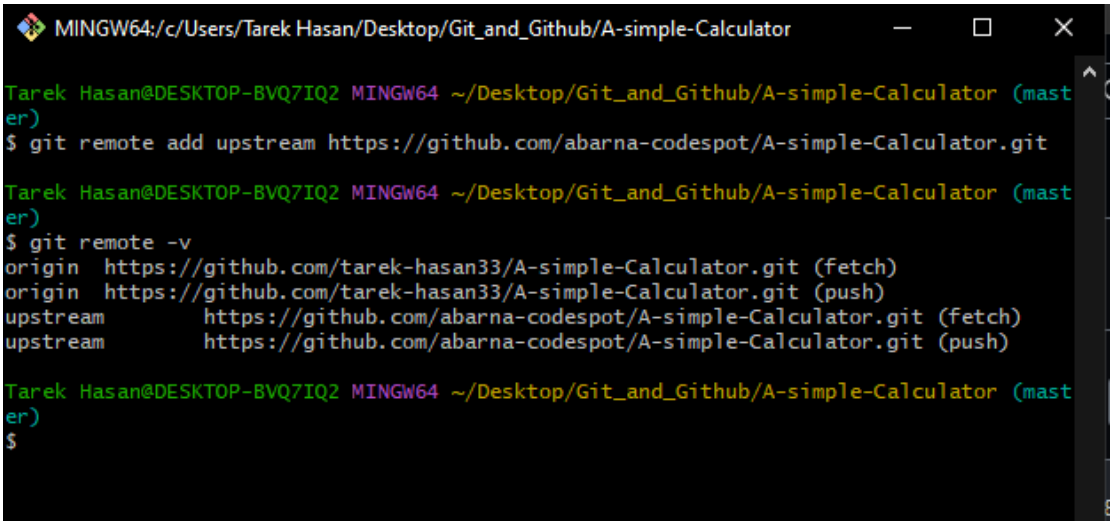


Here you can see the Sync fork button. As well as the status of the branch. If author commits or changed something in the main repo then we will see that our branch is 1 commit behind the main repo. Then we need to sync the fork. After clicking the sync fork button you will have the latest commits in you forked repo. Now we have the updated GitHub repository. But what about the local repository? Yes we need to cover up that too. How? Let's see:

At first let's see how many remotes we have:



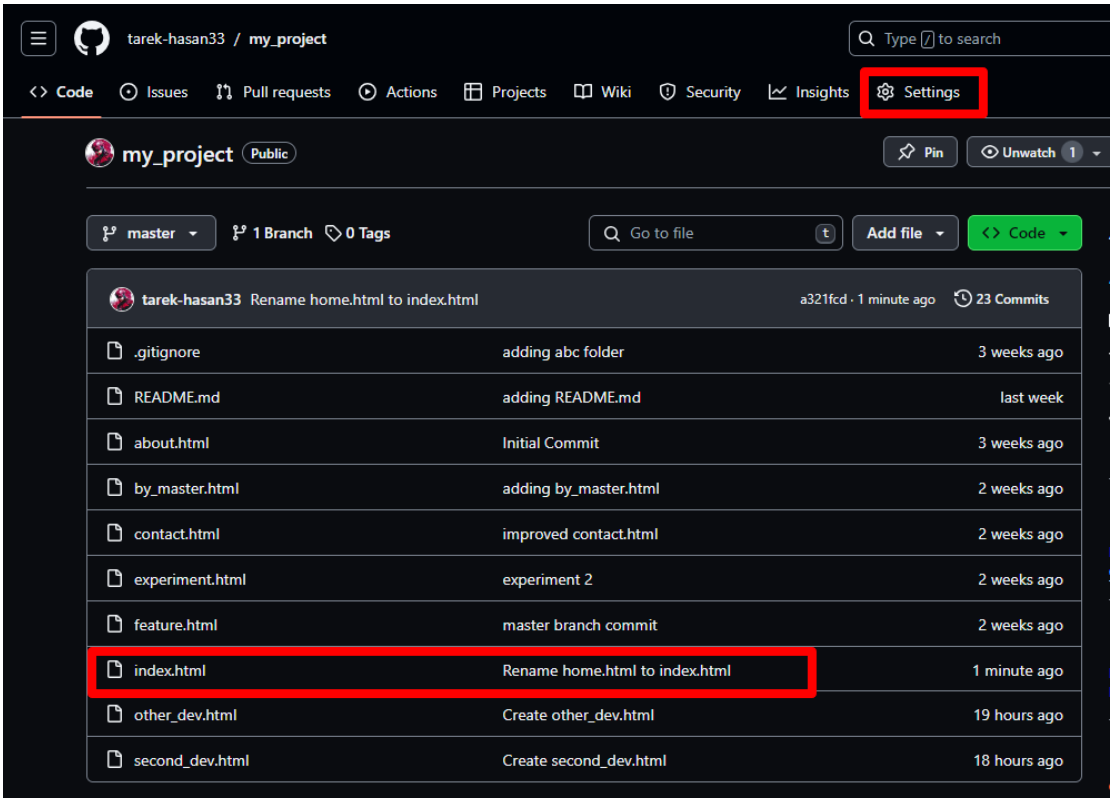
We have this remotes as of now which is for the forked repo in our own github account as you can see. Let's add another remote called upstream. But this remote should be for the main repository. I mean the main repository from which we forked the project. Let's see:



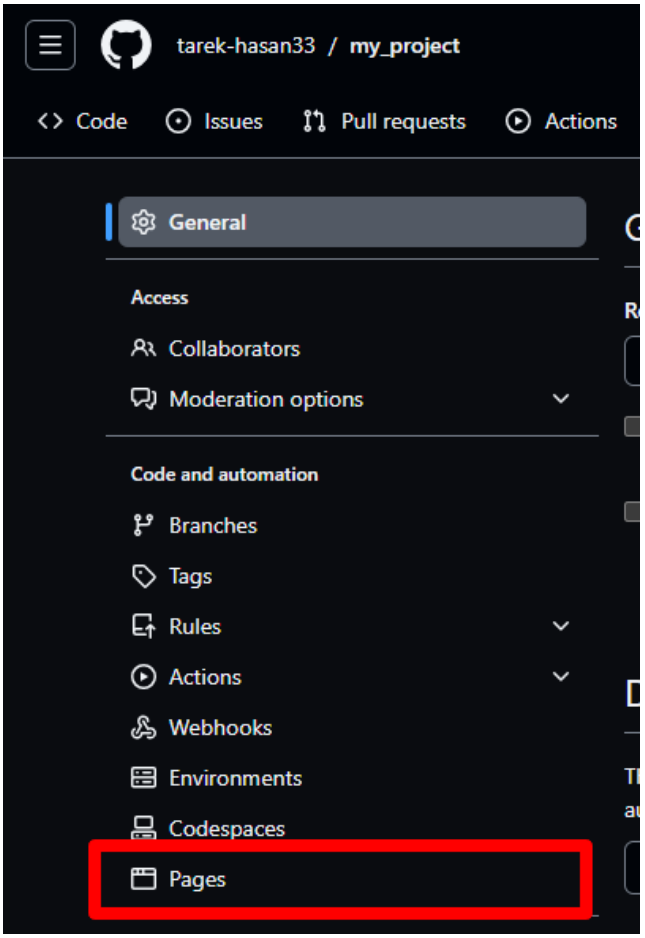
Now we have the remote. We can just fetch and pull from the upstream remote. We can not push as the repo is not ours. Now we can use commands like **git pull upstream master** or **git fetch upstream master** to pull the commits in the local repo.

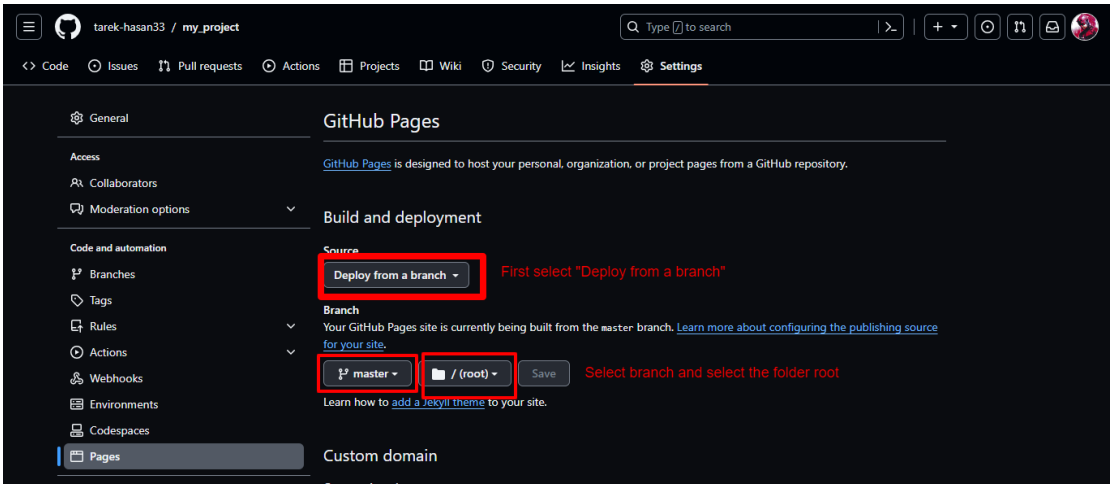
▼ Pages

▶️ There is a thing called GitHub pages. So what is GitHub pages. Well it kind of works like a website hosting function. Suppose you have the repository of you website that you have created on github. You can host it using GitHub pages. You can only host the static website using GitHub pages not dynamic ones. As we already know we have a website in the my_project repo. At first we need to change the home.html to index.html cause GitHub recognized that the index.html file is the home file.



Then we need to go to the setting option. There we will see the pages option:

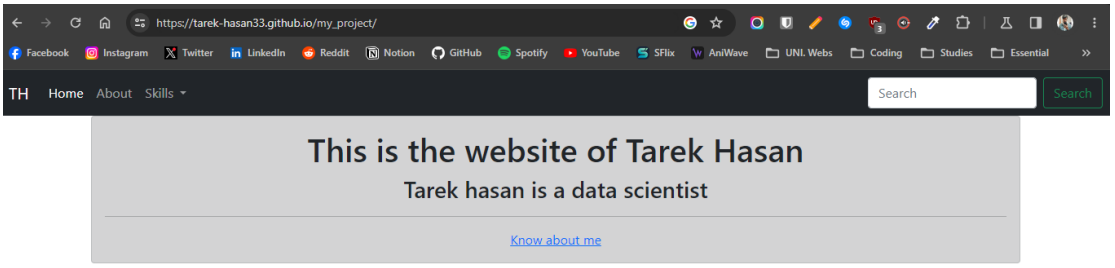




After that you website will be ready. How can you visit your website? Like this:

> 📌 USERNAME.github.io/PROJECTNAME

In our case it is tarek-hasan33.github.io/my_project. Let's visit the website:



As you can see it has published our website.

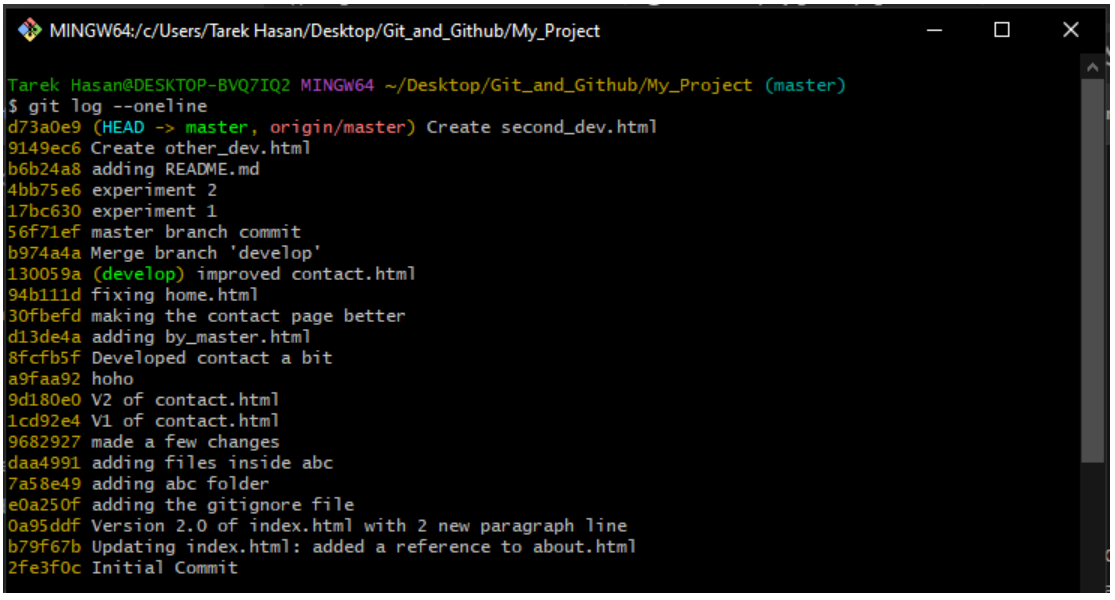
▼ Advance Git Commands

▼ Show me the Commit Changes

▶️ Suppose you have a project in your local machine or in Github and you cloned it. Now you want to see who committed what in the project. Means you want to see in which commits what has been changed. We can do that using this command:

> 📌 git show <COMMIT ID>

How can we get the commit id? By using git log. Let's look at the example:



These are the git logs in oneline for the my_project repo with commit messages. Now suppose I want to check the changed of the commit which has the message **“V2 of contact.html”**

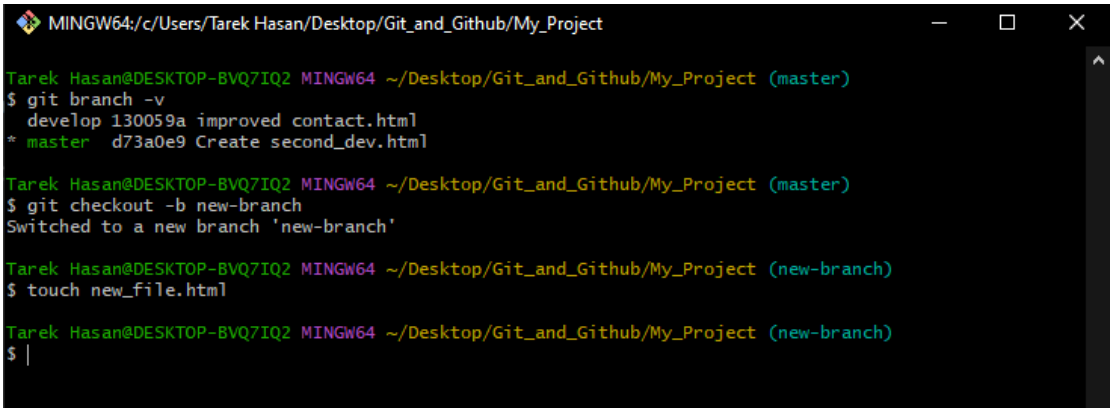
Let’s look at the output:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git show 9d180e0
commit 9d180e0e48d04518ae1645fd900b0d331e09cd64
Author: Tarek Hasan <t.r.hasan33@gmail.com>
Date:   Mon Dec 11 03:01:14 2023 +0600

    V2 of contact.html

diff --git a/contact.html b/contact.html
index 63921a1..a283c4a 100644
--- a/contact.html
+++ b/contact.html
@@ -6,13 +6,8 @@
     <title>Contact Me</title>
 </head>
 <body>
-    <h1>Contact Me</h1>
-
-    <form action="">
-        <input type="text" placeholder="Enter your name">
-        <input type="email" placeholder="Enter you email">
-        <input type="submit" value="Submit">
-    </form>
-
+    skald;fkhalsdhf
+    lakhdflahsdfljhas
+    adlskfhasldhf
 </body>
 </html>
\ No newline at end of file
```

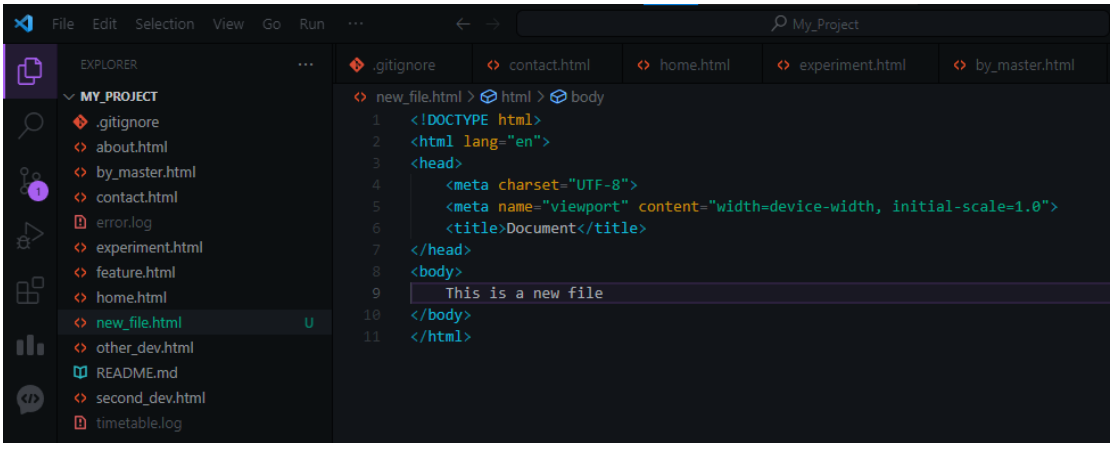
As you can see it shows me the details of that commit like what have I deleted and what have I added in that commit.

▼ Git Stash

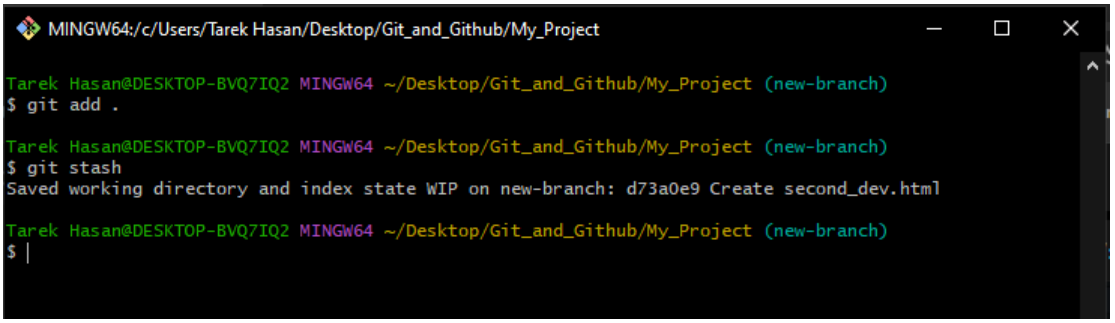
▶️ Suppose your company has a website and you are the one who is managing it. Now you are working on a sperate branch to add a new feature to the website. But you manager calls and asks you to fix a bug in the home page but you’re in the middle of the feature creation process. You don’t want to commit the changes either. If you now visit the master branch you’ll loose the work that you have been doing creating the feature. So what can you do? You can simply stash the file and visit the master branch. It will save the work for you without committing. Let’s see.



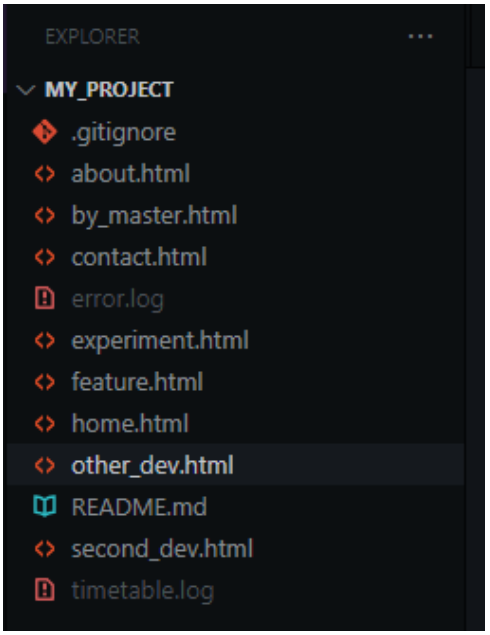
We have created a new branch and also a new file in that branch. Let's edit the file.



Now suppose we have to shift to the master branch without loosing and committing the changes in this new branch. How can we do that? By adding the file to the staging area and stashing it. Let's see:



As you can see it says it saved the file and we have moved back to the last commit. Also we won't be able to see the file:

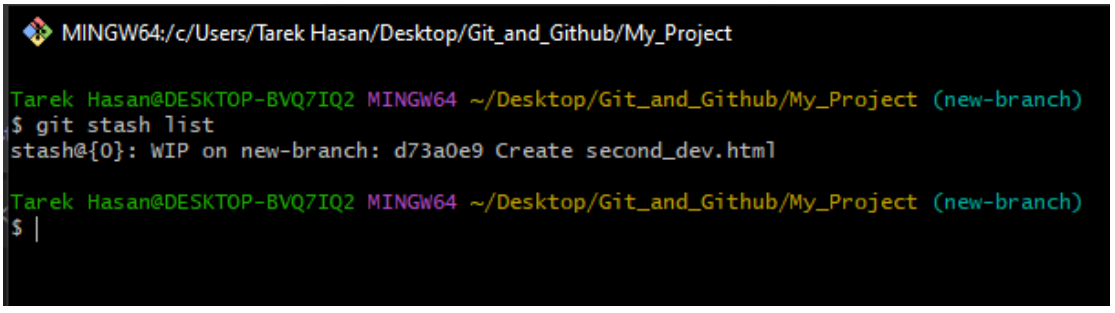


We don’t have the file anymore.

▼ Applying Stash

▶️ In our previous section we have stashed a file. Now how do we un-stash the stashed file? At first let’s see the stashed file list using this command:

> 📌 git stash list



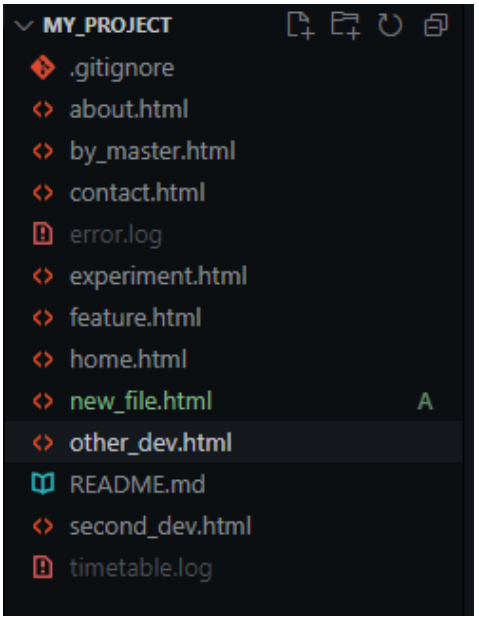
To re-apply the file or you can say restore the file we can use:

> 📌 git stash apply stash@{NUMBER OF THE FILE}

I our case it is stash@{0}. So let’s run the command and see:

```
MINGW64:/c/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git stash apply stash@{0}
On branch new-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   new_file.html

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$
```

It show the file has been restored. Let's verify it from VS Code:



As you can see we have the new_file.html in the repo. But we still have the stash file right? How to delete the stash? We can do that by:

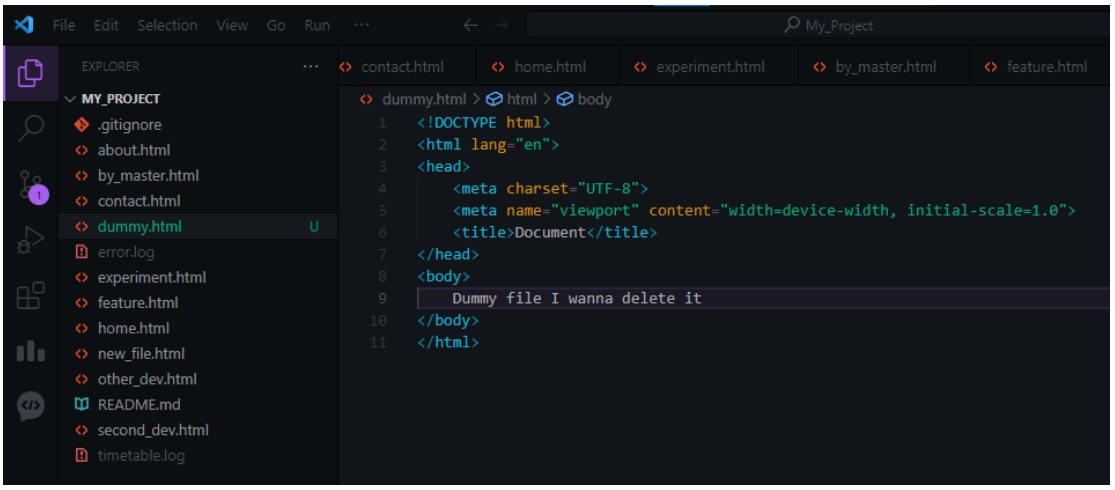
📌 git stash drop stash@{NUMBER OF THE FILE}

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git stash drop stash@{0}
Dropped stash@{0} (cd9bcb217393cc6d77e26511e4d37c3ef64b5b44)

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git stash list

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$
```

As you can see we don't have any stash file remaining.

▼ Cleaning Working Repository

▶️ We are in the new_branch now. So what if we don't want to commit the changes or stash it. What if we just want to clean the working repo and switch to the master repo? We can do that too. It will delete all the changed after the last commit. Let's create a dummy.html in the new_branch.

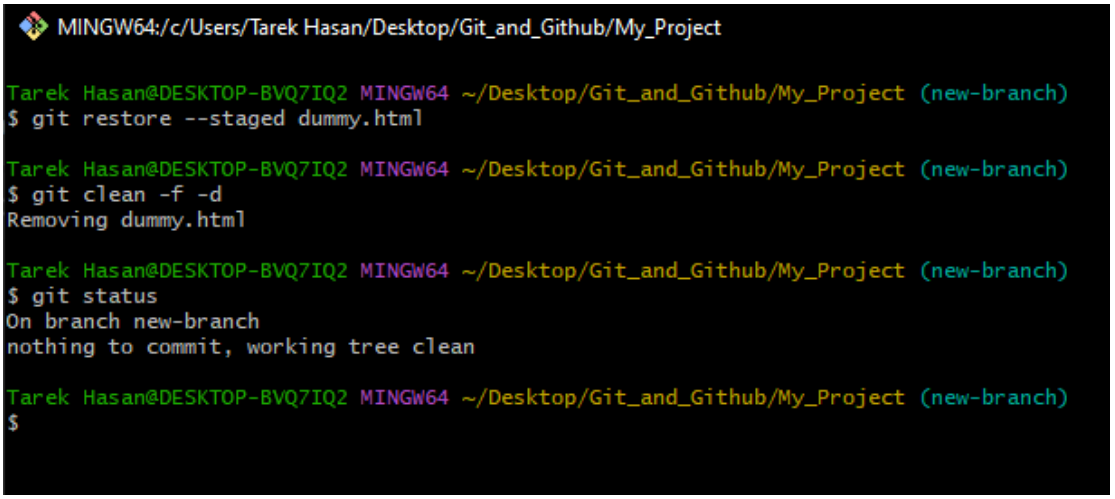


We have this file in the staging area suppose. So if we want to clean the whole repo we can use:

> 📌 git clean -f -d

Let's see:

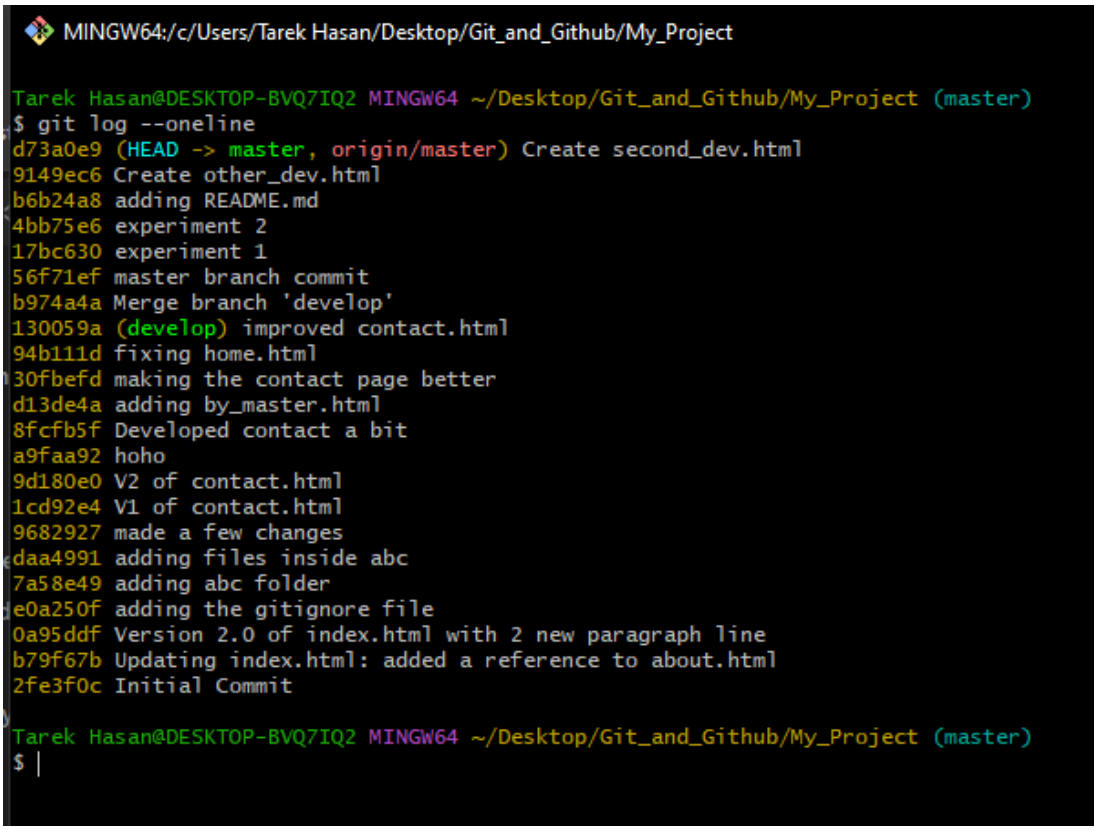
But before running this command we need to remove the file from the staging area.



It has been cleaned.

▼ Changing the Previous Commit Message

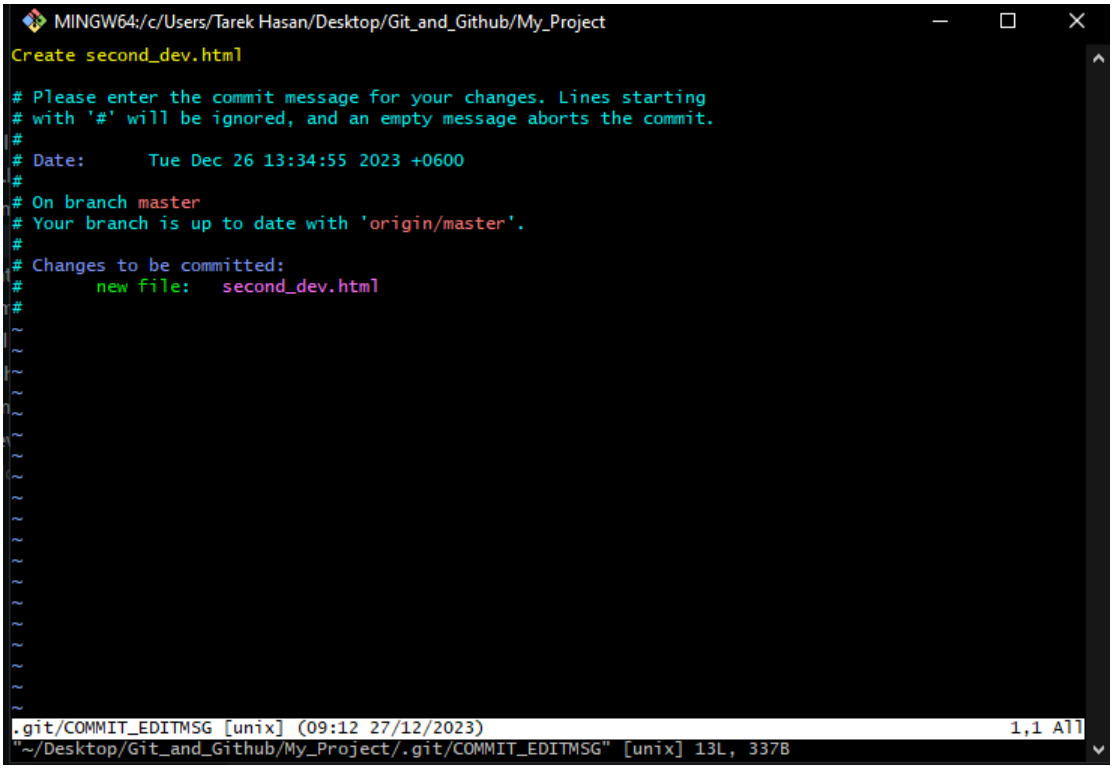
▶️ Let’s look at the git log:



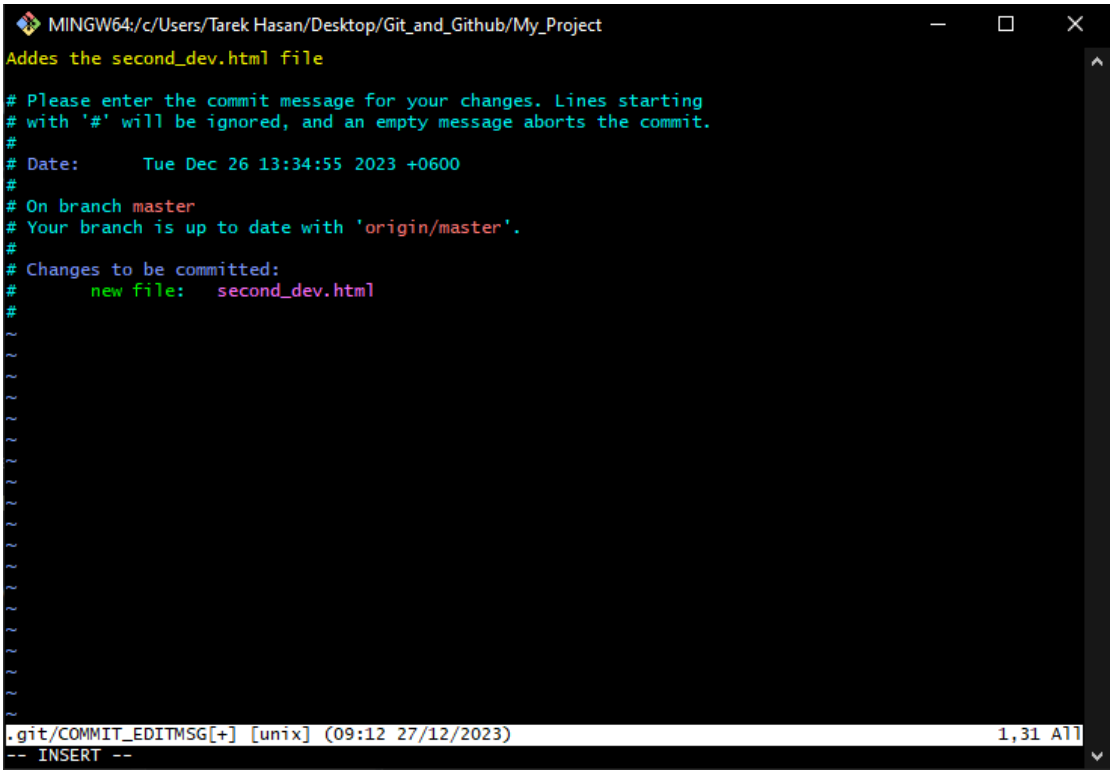
This is our log. Now suppose I want to change the message of the lasted commit. To do that we can use this command:

> 📌 git commit —amend

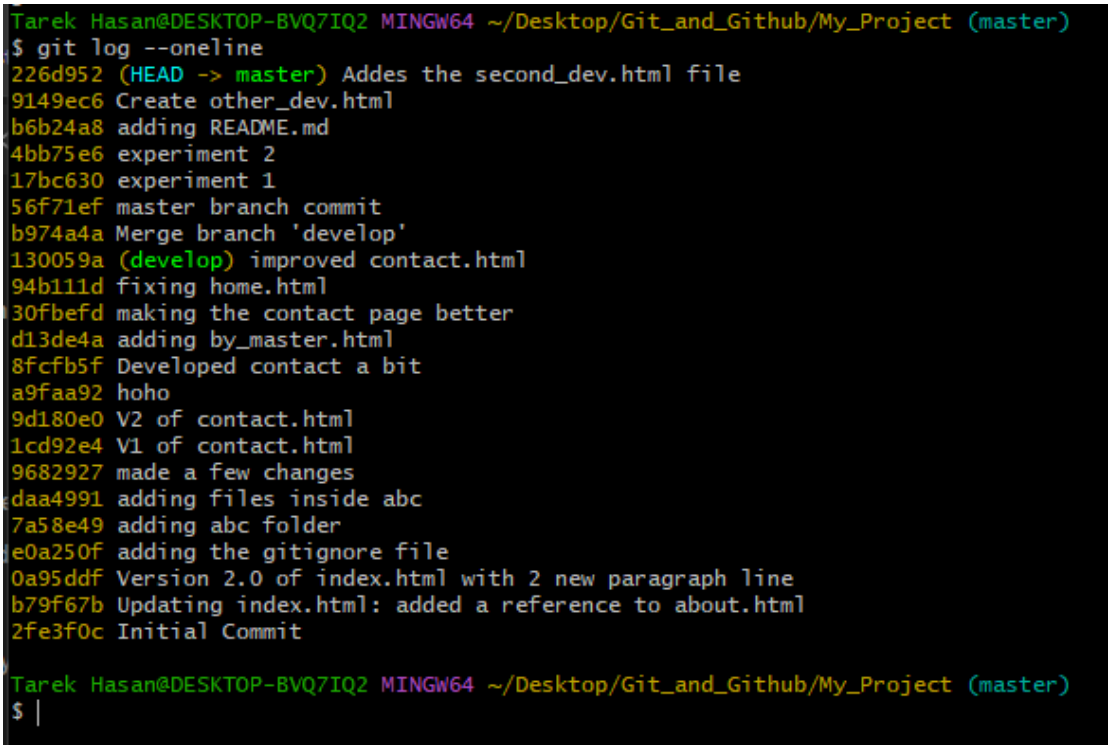
Let’s look into it:



It will show a windows like this. We can now press **[I]** to get to the insert mode and change the message.



Here I have changed the message. Now press [**esq button + : + wq**] to write and quit. Now let's look into the git log:



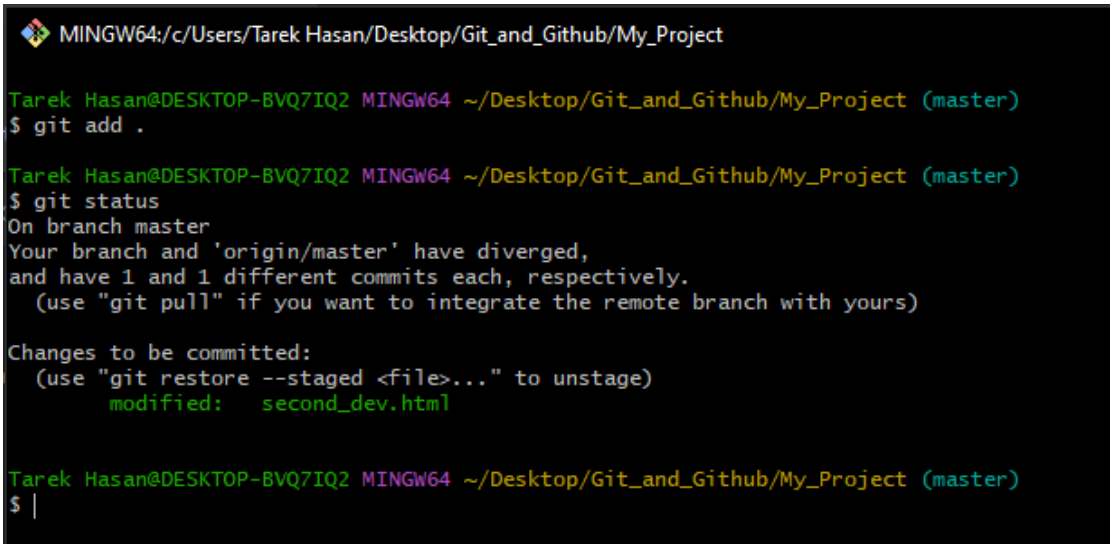
As you can see we have changed the commit message.

▼ Changing Committed Content in the Previous Commit

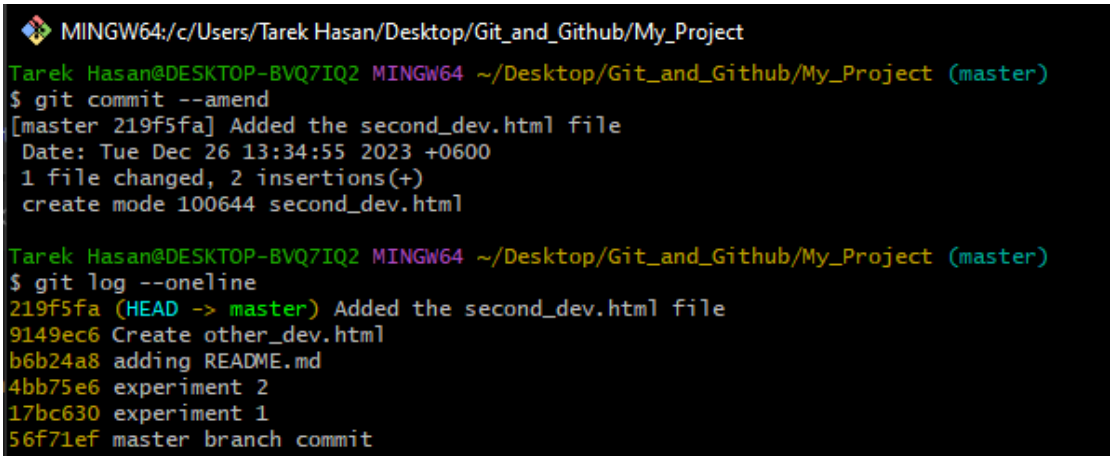
▶️ Suppose we are working with the second_dev.html file. Do change the content of this file while keeping the same commit we need to follow these steps:

> 📌 1. Make Changed in the Working Directory
> 2. Add the files to the staging area
> 3. Re-run the command
> **git commit —amend**

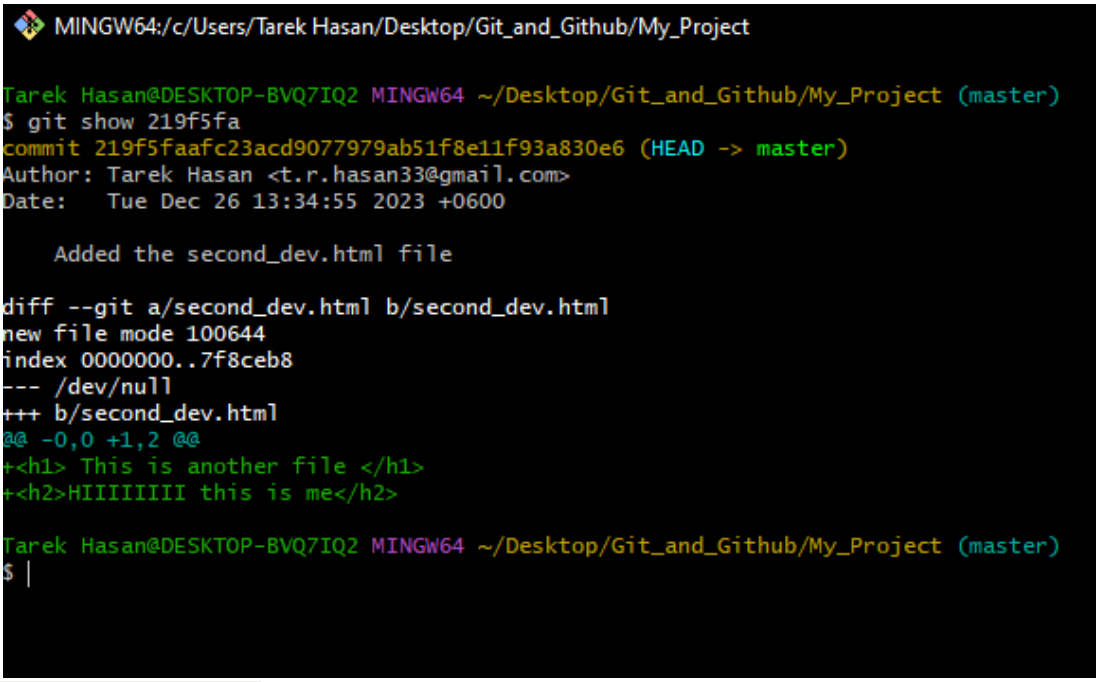
Let’s edit the second_dev.html file and add it to the staging area.



Let’s re-run the command:



It still shows the same commit but let’s see the commit changes:



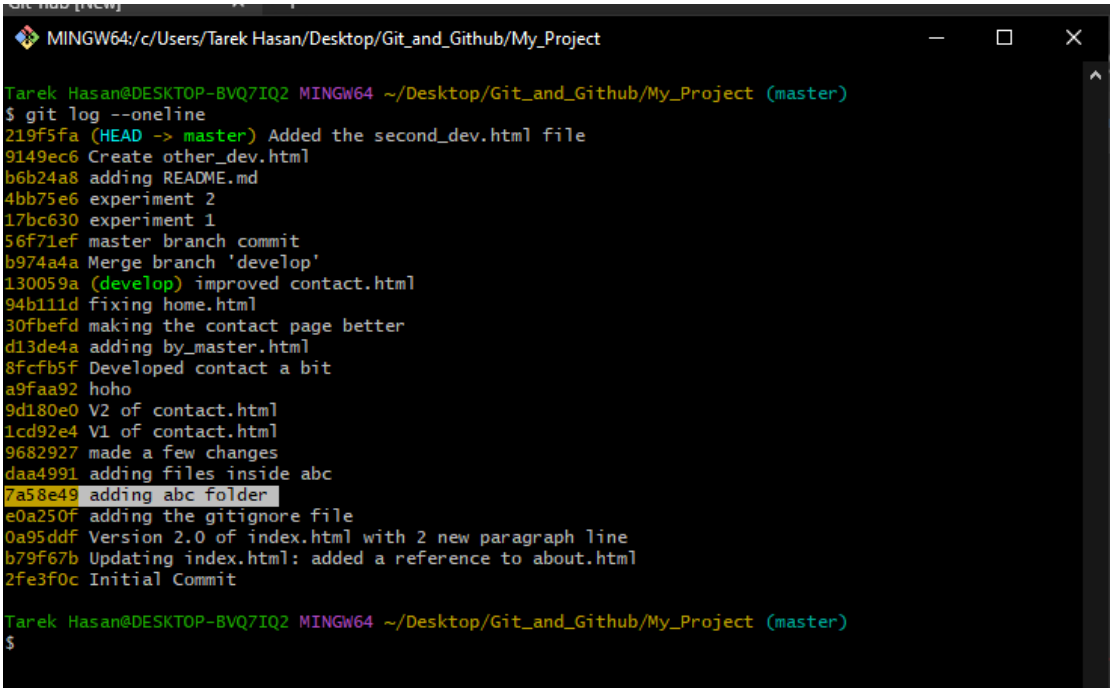
▼ Going to The Past

▼ Travelling to the Past Commits

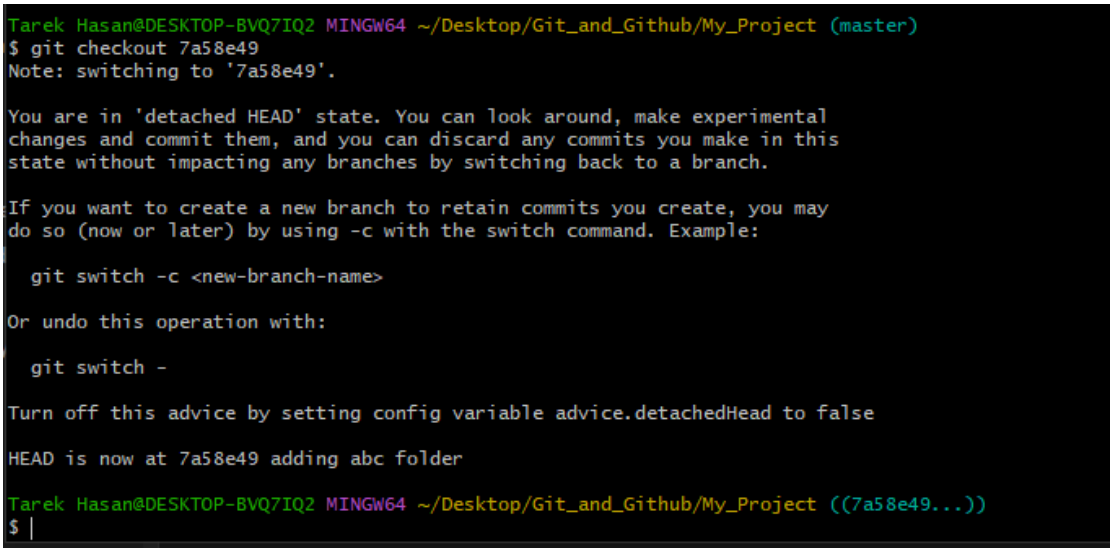
▶️ Suppose you want to look into a commit which is like 20 commits older. You had a important file in that commit so now you want look into that file. How can you do that? By this command:

> 📌 git checkout <COMMIT ID>

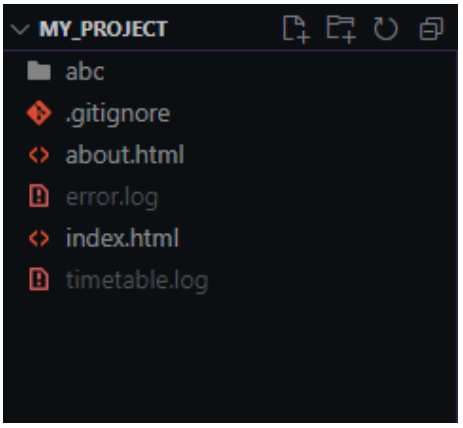
Let’s look into the git log:



Suppose we want to visit the selected commit. What will happen if we run the command? Well it will go back to that past stage means it will not show the current files. It will just show those old files. Let's see:



We have switched to that commit. Let's look into the VS Code.



We don't have that many files right? What does this command do actually? It moves the head from current commit to the commit we are checking out. But can we move back to the recent commit or master branch? Yes of course. To do that we can run this command:

> 📌 git switch -

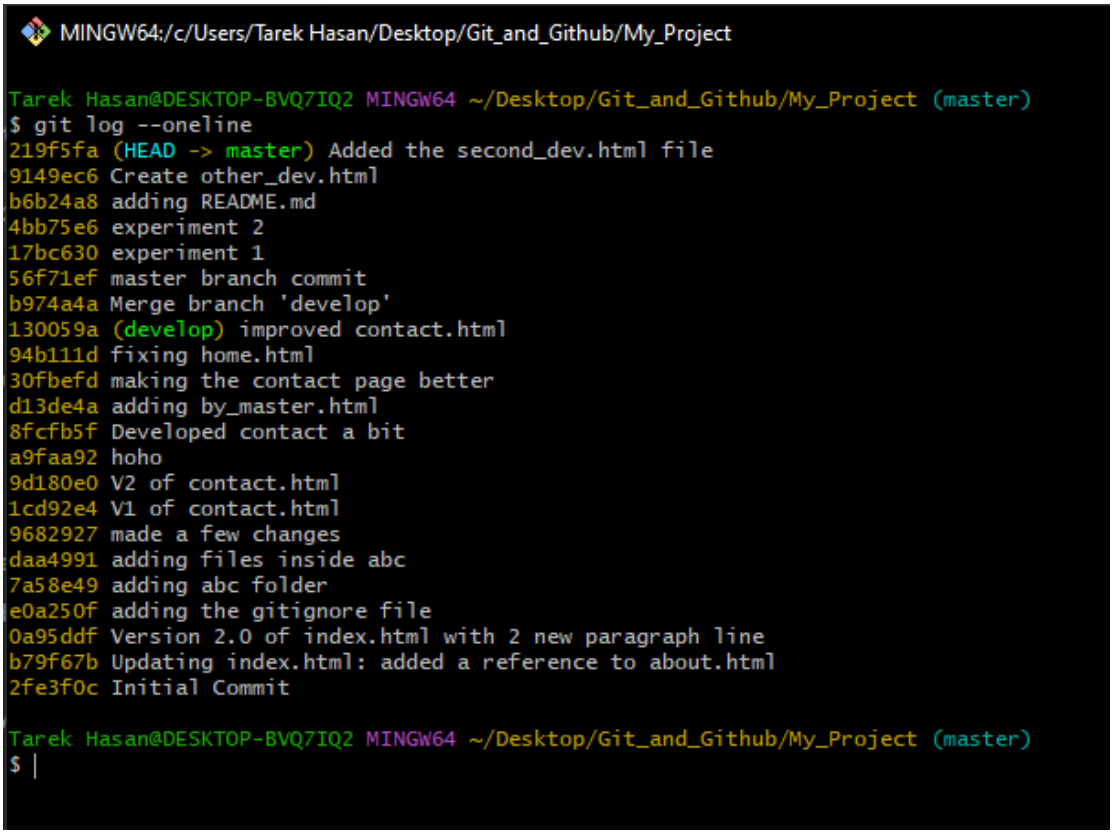
Let's try this out:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project ((7a58e49...))
$ git switch -
Previous HEAD position was 7a58e49 adding abc folder
Switched to branch 'master'
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

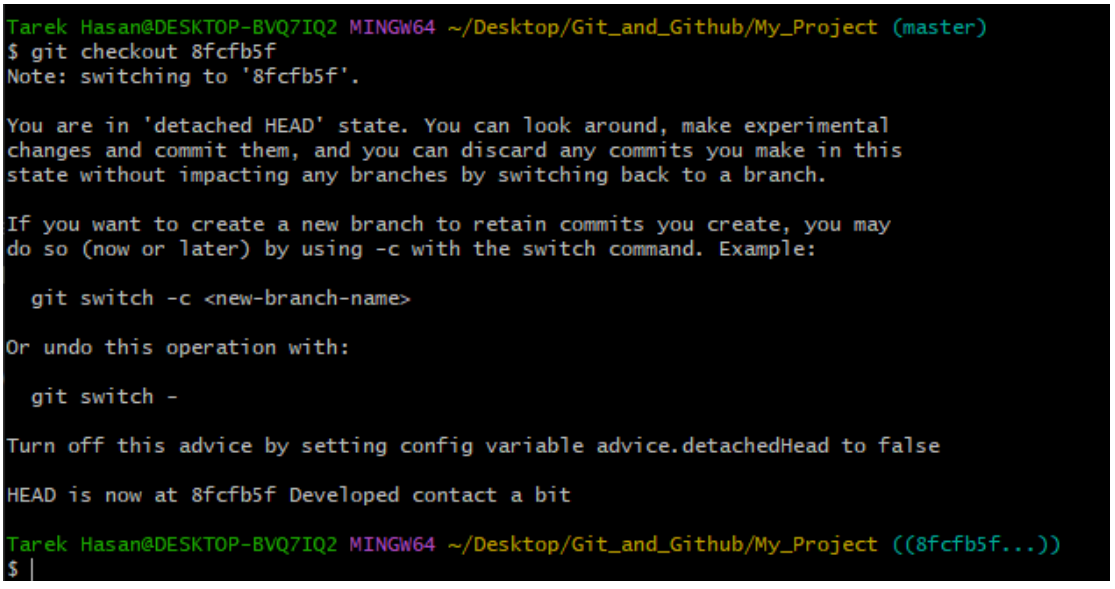
It now has been switched to master branch. But what if you wanted to move to the older commit just so that you can create a new feature from that moment? If you commit at that moment you might loose all the things you had after that commit. Means you will be starting from that commit. To overcome that we can create a new branch from that commit. Let's move to the next chapter.

▼ Making Branches from Old Commits

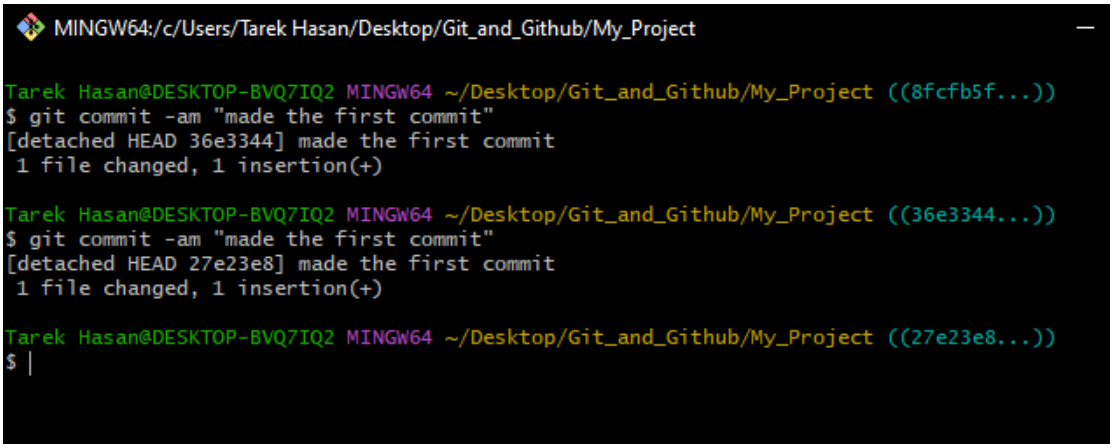
▶️ Let's look into the git log:



Suppose we want to checkout “Developed contact a bit” commit. Let’s check that out:



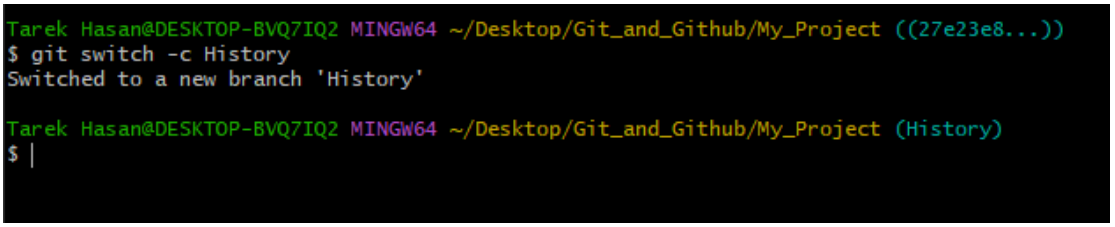
We are in that commit now. Let’s edit some files and make 2 commit.



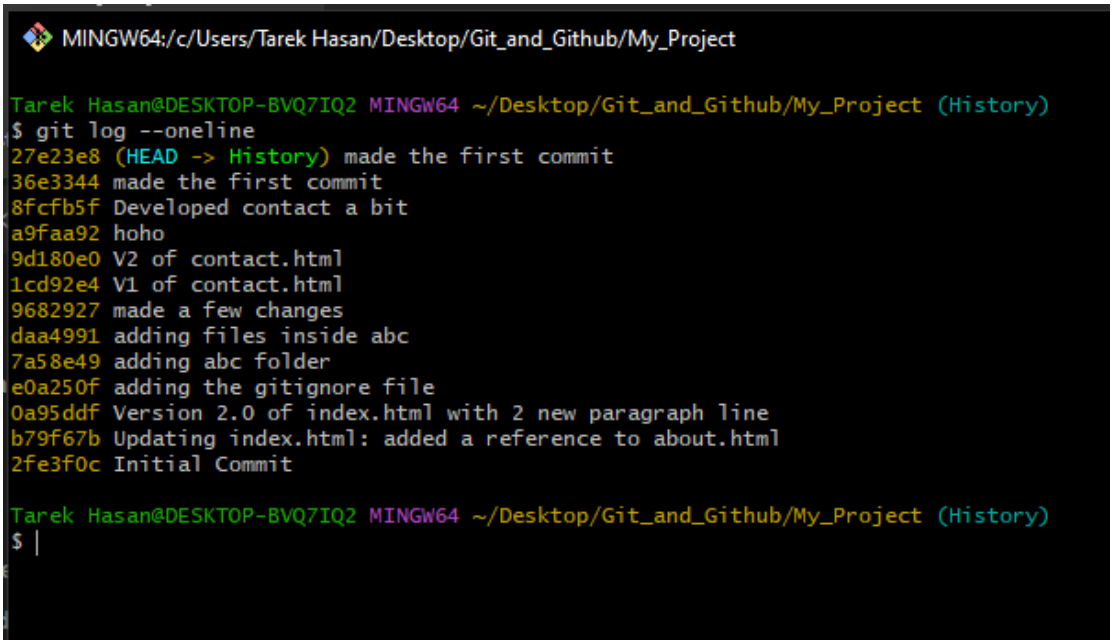
I have made 2 commits now. Now what if we want to switch to the master branch? We can't. So we need to create a new branch from this point using this command:

📌 git switch -c <BRANCH NAME>

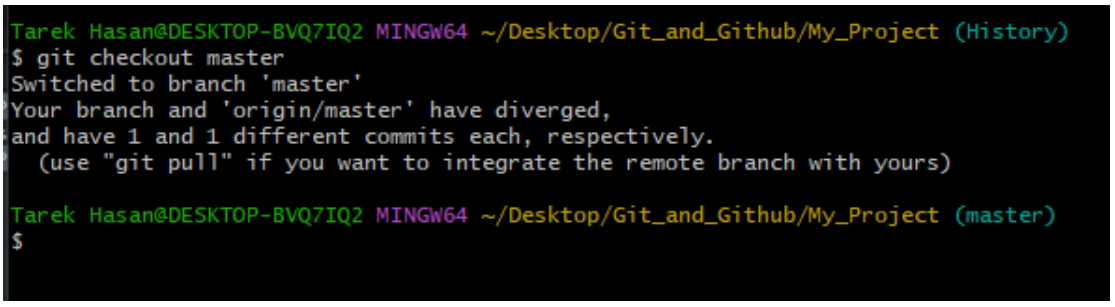
Let's practically do this:



Now the changes are in a different branch from that commit. Let's try to switch to master branch. But first we should look into the git log:



As you can see we don't have too much commits cause we only have the commits from that point where we created the branch. Let's switch to master branch now:



▼ Git Revert

▶️ What does git revert does? If you want to revert the changes of a commit then you can use the git revert command. Suppose you added 6 lines to Readme.md in this [91ufu2] commit. If you use the command it will delete these 6 lines. So the command is:

> 📌 git revert <COMMIT ID>

Let's look at the practical example:

```
MINGW64:/c/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git log --oneline
219f5fa (HEAD -> master) Added the second_dev.html file
9149ec6 Create other_dev.html
b6b24a8 adding README.md
4bb75e6 experiment 2
17bc630 experiment 1
56f71ef master branch commit
b974a4a Merge branch 'develop'
130059a (develop) improved contact.html
94b111d fixing home.html
30fbefd making the contact page better
d13de4a adding by_master.html
8fcfb5f Developed contact a bit
a9faa92 hoho
9d180e0 V2 of contact.html
1cd92e4 V1 of contact.html
9682927 made a few changes
daa4991 adding files inside abc
7a58e49 adding abc folder
e0a250f adding the gitignore file
0a95ddf Version 2.0 of index.html with 2 new paragraph line
b79f67b Updating index.html: added a reference to about.html
2fe3f0c Initial Commit

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

Suppose we want to revert the selected commit. Let's see what we did in that commit:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git show d13de4a
commit d13de4a8508cad17b387d9867bdcdd79e11175aa
Author: Tarek Hasan <t.r.hasan33@gmail.com>
Date:   Wed Dec 13 18:25:36 2023 +0600

    adding by_master.html

diff --git a/by_master.html b/by_master.html
new file mode 100644
index 0000000..93cd263
--- /dev/null
+++ b/by_master.html
@@ -0,0 +1,12 @@
+<!DOCTYPE html>
+<html lang="en">
+<head>
+    <meta charset="UTF-8">
+    <meta name="viewport" content="width=device-width, initial-scale=1.0">
+    <title>Master</title>
+</head>
+<body>
+    <h1>By master</h1>
+
+</body>
+</html>
\ No newline at end of file

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

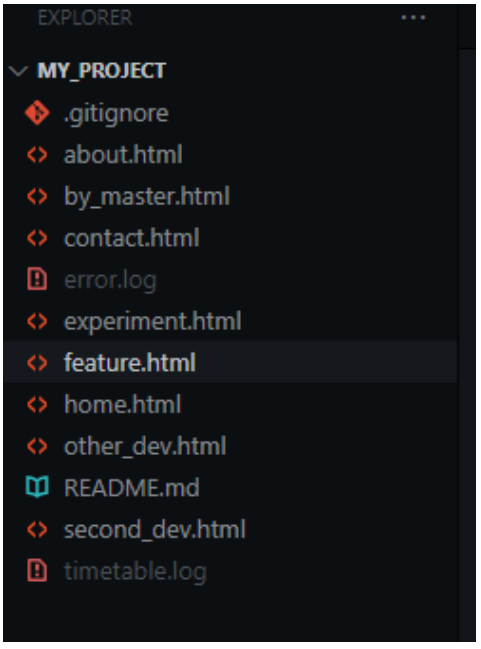
We added the by_master.html file in that comment. So what will happen if we revert this commit? Yes the file will disappear. To do that we can use this command:

> 📌 git revert <COMMIT ID>

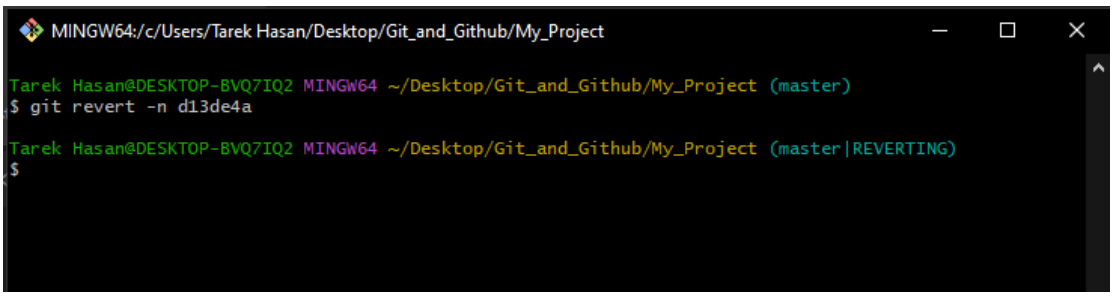
But this command will revert the commit as well as commit the changes. That means if we run this commit using that commit id we will loose this file. But there is a flag which we can use to revert the changed but it will not commit at sudden. The command is:

> 📌 git revert -n <COMMIT ID>

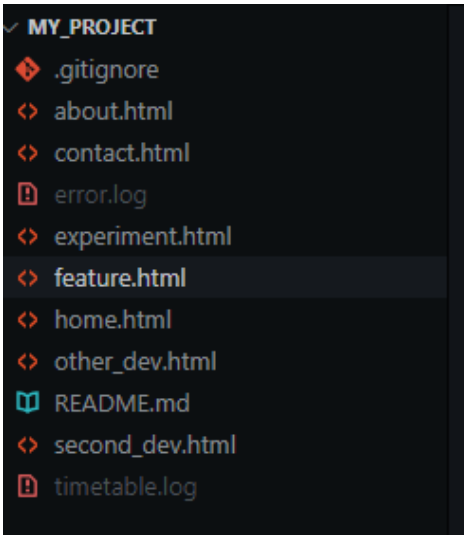
Let's try this command. But first we should look into the VS Code:



We have by_master.html in our repo. Let's run the command:



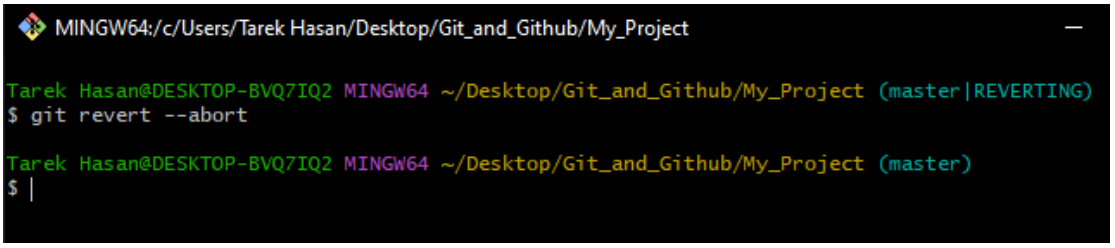
Let’s look into the VS Code:



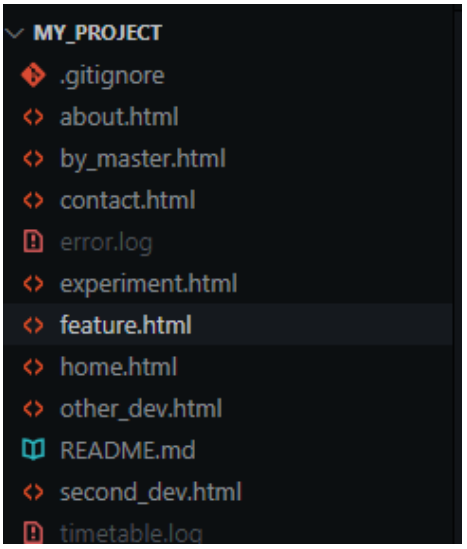
We don’t have that file anymore. But we can abort the changes too. To do that we can use:

> 📌 git revert —abort

Let’s run this command:



Now let’s look into the VS Code window:



We have the by_master.html now.

▼ **Git Reset [CAUTION]**

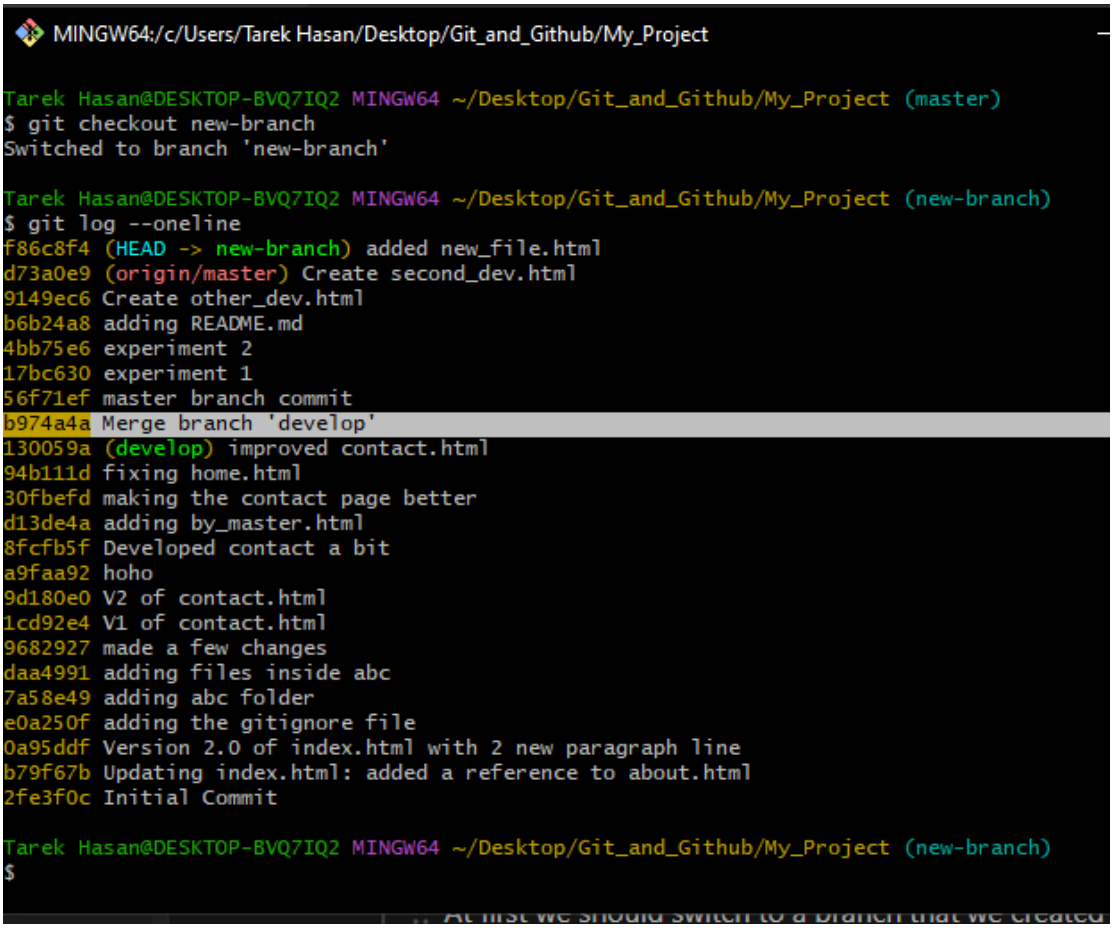
▶️ Let's see what this command does. There are 2 commands for reset:

> 📌 git reset —soft <COMMIT ID> [Though you don't have to put —soft as it's by default]

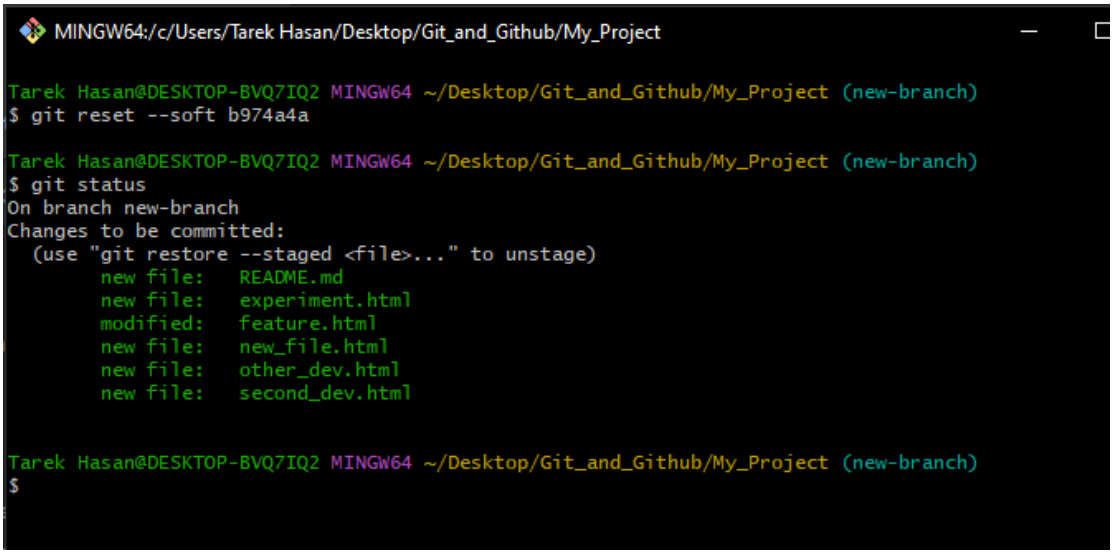
> 📌 git reset —hard <COMMIT ID>

When we use the —soft command it deletes all the commit till the commit id we have given. But it will give you the rights to delete. It will just remove them and it's up to you if you want to delete them permanently or not. Let's use the —soft command:

At first we should switch to a branch that we created earlier named new_branch.



Suppose we want to delete till the commit we have selected. Let's run the command:



As you can see the —soft command is giving me chance to restore them instead of deleting them. So how should we delete them? We can just clean the repo.

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch
$ git log --oneline
b974a4a (HEAD -> new-branch) Merge branch 'develop'
130059a (develop) improved contact.html
94b111d fixing home.html
30fbefd making the contact page better
d13de4a adding by_master.html
8fcfb5f Developed contact a bit
a9faa92 hoho
9d180e0 V2 of contact.html
1cd92e4 V1 of contact.html
9682927 made a few changes
daa4991 adding files inside abc
7a58e49 adding abc folder
e0a250f adding the gitignore file
0a95ddf Version 2.0 of index.html with 2 new paragraph line
b79f67b Updating index.html: added a reference to about.html
2fe3f0c Initial Commit
```

As you can see we don't have the commits after the commit we have selected. Now what about the —hard command? Let's see:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git reset --hard 9d180e0
HEAD is now at 9d180e0 V2 of contact.html

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git status
On branch new-branch
nothing to commit, working tree clean

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$ git log --oneline
9d180e0 (HEAD -> new-branch) V2 of contact.html
1cd92e4 V1 of contact.html
9682927 made a few changes
daa4991 adding files inside abc
7a58e49 adding abc folder
e0a250f adding the gitignore file
0a95ddf Version 2.0 of index.html with 2 new paragraph line
b79f67b Updating index.html: added a reference to about.html
2fe3f0c Initial Commit

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (new-branch)
$
```

As you can see we have deleted everything till then.

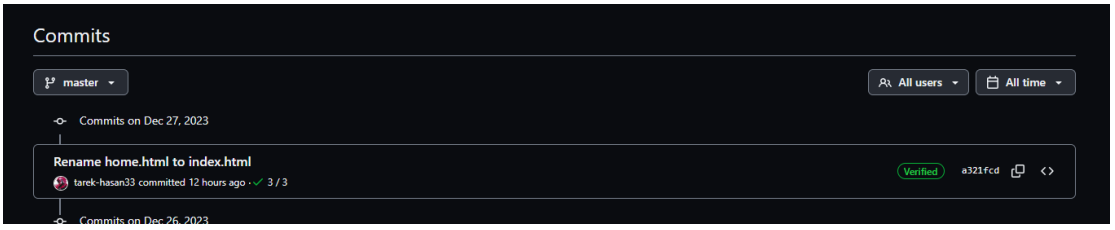
▼ Deleting a Commit from GitHub

▶️ So what exactly does it mean by deleting a commit from GitHub? Well till now we already know that we cannot delete a commit from the middle. We have delete a whole part of commit to reach to that point of commit. In this case it is the same. We will have to delete a commit and forcefully push to commit to the central repo I mean github. That's how we are going to delete a commit from GitHub. So let's see:

```
MINGW64:/c/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git branch
  History
  develop
* master
  new-branch

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

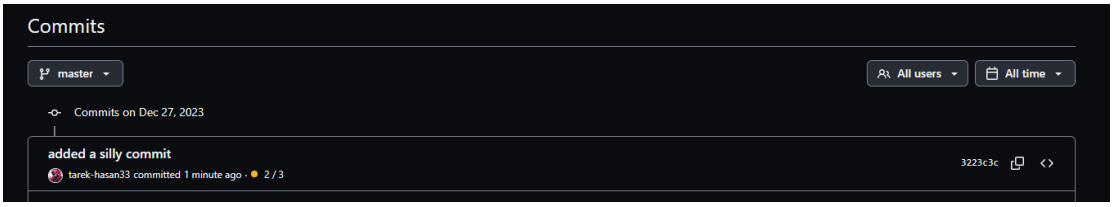
As you can see we are in the master branch. Let's look at the git log and GitHub log:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git log --oneline
a321fcd (HEAD -> master, origin/master) Rename home.html to index.html
d73a0e9 Create second_dev.html
9149ec6 Create other_dev.html
b6b24a8 adding README.md
4bb75e6 experiment 2
17bc630 experiment 1
56f71ef master branch commit
b974a4a Merge branch 'develop'
130059a (develop) improved contact.html
94b111d fixing home.html
30fbefd making the contact page better
d13de4a adding by_master.html
8fcfb5f Developed contact a bit
a9faa92 hoho
9d180e0 (new-branch) V2 of contact.html
1cd92e4 V1 of contact.html
9682927 made a few changes
daa4991 adding files inside abc
7a58e49 adding abc folder
e0a250f adding the gitignore file
0a95ddf Version 2.0 of index.html with 2 new paragraph line
```



As you can see both git and github has the same commit. Let's change something in the file and commit and push it to the github. Then we will try do delete that commit from the github.

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git commit -am "added a silly commit"
[master 3223c3c] added a silly commit
 1 file changed, 1 insertion(+)

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 319 bytes | 319.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/tarek-hasan33/my_project.git
   a321fcd..3223c3c  master -> master

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git log --oneline
3223c3c (HEAD -> master, origin/master) added a silly commit
a321fcd Rename home.html to index.html
d73a0e9 Create second_dev.html
9149ec6 Create other_dev.html
b6b24a8 adding README.md
```



Now both git and github has the same commit. What do we need to do now in order to delete a commit from github? We first need to delete the commit from the git using hard reset then we need to force push the repo to github. Let’s do this:

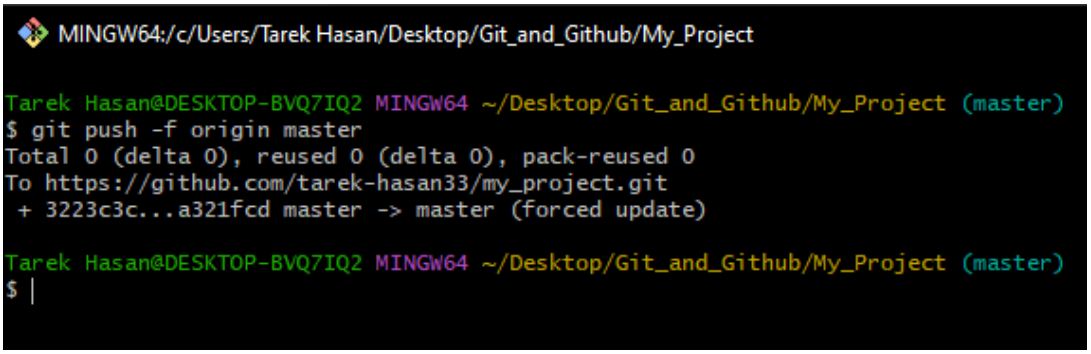
```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git reset --hard a321fcd
HEAD is now at a321fcd Rename home.html to index.html

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git log --oneline
a321fcd (HEAD -> master) Rename home.html to index.html
d73a0e9 Create second_dev.html
9149ec6 Create other_dev.html
b6b24a8 adding README.md
4bb75e6 experiment 2
17bc630 experiment 1
56f71ef master branch commit
```

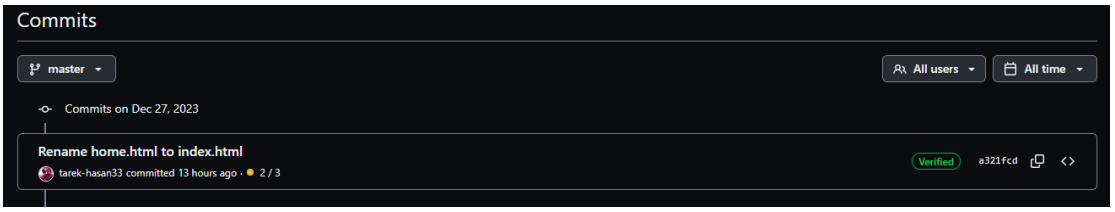
As you can see we don’t have the silly commit anymore. So how can we force push it to the github? Using this command:

> 📌 git push -f <REMOTE NAME> <BRANCH NAME>

In our case the command should be: **git push -f origin master**

Let’s run the command and see what has changed in the github:

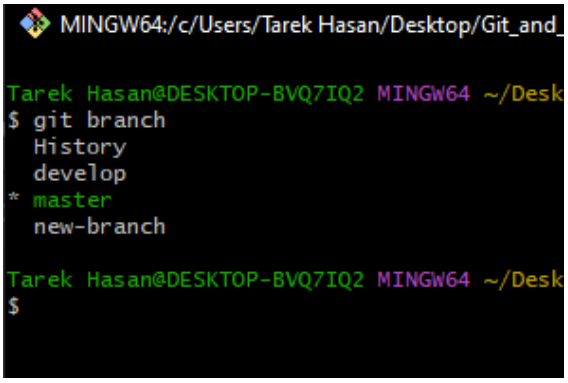




Yes we have moved back to the last commit.

▼ Cherry-Picking Commits

▶️ What does cherry picking commits means? Let’s see how many branches we have:



We have 4 branches. Let’s just say for now that History branch has few commit. Let’s see:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git checkout history
Switched to branch 'history'

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (history)
$ git log --oneline
27e23e8 (HEAD, History) made the first commit
36e3344 made the first commit
8fcfb5f Developed contact a bit
a9faa92 hoho
9d180e0 (new-branch) V2 of contact.html
1cd92e4 V1 of contact.html
9682927 made a few changes
daa4991 adding files inside abc
7a58e49 adding abc folder
e0a250f adding the gitignore file
0a95ddf Version 2.0 of index.html with 2 new paragraph line
b79f67b Updating index.html: added a reference to about.html
2fe3f0c Initial Commit

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (history)
$
```

Suppose we want to pick the commit with **COMMIT ID: 36e3344 with the commit message "made the first commit"** to the master branch and merge it. I mean we can merge the whole History branch with the master but we don't want that. We just want to merge that specific commit to the master branch. We can use the cherry pick command to do that. How?

Let's see what we have changed in that commit:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (history)
$ git show 36e3344
commit 36e3344f1c987f254798b174ad6e8c295b0eb7e9
Author: Tarek Hasan <t.r.hasan33@gmail.com>
Date:   Wed Dec 27 12:01:04 2023 +0600

    made the first commit

diff --git a/feature.html b/feature.html
index 2ac641d..1c78bb3 100644
--- a/feature.html
+++ b/feature.html
@@ -7,5 +7,6 @@
 </head>
 <body>
     <h1>This is a new feature</h1>
+    <h2>This is another feature</h2>
 </body>
 </html>
\ No newline at end of file

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (history)
$
```

It says we have added a line to **feature.html** file which is **"This is another feature".**

Let's now shift to the master branch and look into the feature.html file if we have that line or not:

```
MINGW64:/c/Users/Tarek Hasan/Desktop/Git_and_Github/My_Project

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (history)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$
```

contact.html | feature.html | index.html

feature.html > html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>This is a new feature</h1>
    this
</body>
</html>
```

As you can see we don't have the line in the **feature.html** which is in the master branch. Let's cherry-pick that commit and see what happens. To do that we can run this command:

📌 git cherry-pick <COMMIT ID>

Let's run this command:

```
Tarek Hasan@DESKTOP-BVQ7IQ2 MINGW64 ~/Desktop/Git_and_Github/My_Project (master)
$ git cherry-pick 36e3344
Auto-merging feature.html
CONFLICT (content): Merge conflict in feature.html
error: could not apply 36e3344... made the first commit
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git cherry-pick --continue".
hint: You can instead skip this commit with "git cherry-pick --skip".
hint: To abort and get back to the state before "git cherry-pick",
hint: run "git cherry-pick --abort".
```

Now we have the changed made in the master branch.

NOTE: I had so many texts after running the command because there was a conflict. It will not be an issue with everyone.