

# INDEX

1. Git and GitHub	01
1.1. Git and Github are different	01
2. Terminology	02
2.1. Repository	02
2.2. Your config setting	03
2.3. Creating a repository	04
2.4. Commit Way	05
2.5. Stage	06
2.6. Commit Command	07
2.7. Logs	08
2.8. Gitignore	09
3. Branched in git	11
3.1. Head in git	11
3.2. Git Graph tool	14
3.3. Merging branches	14
3.3.1. Fast Forward Merge	14
3.3.2. Not Fast Forward Merge	17
3.4. Managing conflicts	18
3.5. Conflicts creation and solving	19
3.6. Rename a branch	22
3.7. Delete a branch	22
3.8. Checkout a branch	23
3.9. List all branches	23

4. Diff stash and tags	24
4.1. Git diff	24
4.1.1. Comparing working Directory and Staging Area	24
4.1.2. Comparing Staging Area with Repository	25
4.1.3. Comparing Between Branches	25
4.1.4. Comparing Specific Commits	25
4.2. Git Stash	26
4.2.1. Naming the stash	26
4.2.2. View the Stash list	26
4.2.3. Apply the stash	27
4.2.4. Apply the Specific Stash	27
4.2.5. Applying and Dropping the Stash	27
4.2.6. Drop the Stash	27
4.2.7. Applying Stash to a Specific branch	28
4.2.8. Clearing the Stash	28
4.3. Git Tags	28
4.3.1. Creating a Tag	28
4.3.2. Creating an annotated tag	29
4.3.3. List all tags	29
4.3.4. Tagging a specific Commit	29
4.3.5. Push tags to remote repository	29
4.3.6. Delete a tag	30
4.3.7. Delete tag on remote repository	30
5. Rebase and reflog	31
5.1. Rabase in git	31

5.2.	Merge commits	31
5.3.	Rebase	32
5.4.	Ensure you are on the branch you want to rebase	32
5.5.	Resolve any conflicts	33
5.6.	Git reflog	33
5.7.	View the reflog	33
5.8.	Find specific commit	33
5.9.	Recover lost commit or changes	34
6.	Started with Github	35
6.1.	Configure your config file	35
6.2.	Setup ssh key and add to github	35
6.2.1.	Step 1: Generate a new SSH Ke5	36
6.2.2.	Save the Key	36
6.2.3.	Add key to your ssh-agent	36
6.2.4.	Add key to github	36
6.3.	Adding code to remote repository	36
6.3.1.	Check remote url setting	37
6.3.2.	Add remote repository	37
6.3.3.	Push code to remote repository	38
6.3.4.	Setup an upstream remote	38
6.4.	Readme.md file	39
6.5.	Get code from remote repository	40
6.5.1.	Fetch Code	40
6.5.2.	Pull Code	41
7.	Open Source	42

## **1. Git and GitHub**

**Git** is a version control system that allows you to track changes to your files and collaborate with others. It is used to manage the history of your code and to merge changes from different branches. I can understand that as of now these terms like version control, branches, and merges are not familiar to you.

### **1.1. Git and Github are different**

Git is a version control system that is used to track changes to your files. It is a free and open-source software that is available for Windows, macOS, and Linux. Remember, GIT is a software and can be installed on your computer.

Github is a web-based hosting service for Git repositories. Github is an online platform that allows you to store and share your code with others. It is a popular platform for developers to collaborate on projects and to share code. It is not that Github is the only provider of Git repositories, but it is one of the most popular ones.

### **ID and PASS**

surajpatel9390@gmail.com  
SURAJ2004@

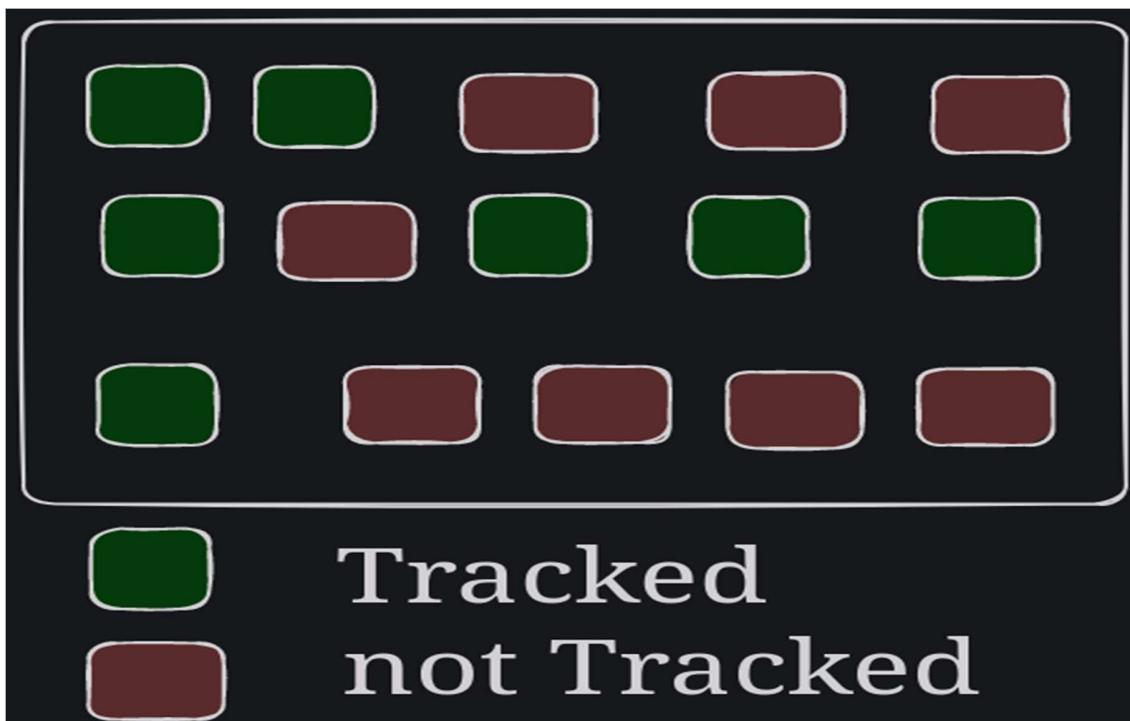
## **2. Terminology**

### **2.1. Repository**

A repository is a collection of files and directories that are stored together. It is a way to store and manage your code. A repository is like a folder on your computer, but it is more than just a folder. It can contain other files, folders, and even other repositories. You can think of a repository as a container that holds all your code.

There is a difference between a software on your system vs tracking a particular folder on your system. At any point you can run the following command to see the current state of your repository:

`git status`



Not all folders are meant to be tracked by git. Here we can see that all green folders are projects are getting tracked by git but red ones are not.

## **2.2. Your config settings**

Github has a lot of settings that you can change. You can change your username, email, and other settings. Whenever you checkpoint your changes, git will add some information about your such as your username and email to the commit. There is a git config file that stores all the settings that you have changed. You can make settings like what editor you would like to use etc. There are some global settings and some repository specific settings.

Let's setup your email and username in this config file.

```
git config --global user.email "your-email@example.com"  
git config --global user.name "Your Name"
```

Now you can check your config settings:

```
git config --list
```

This will show you all the settings that you have changed.

Example:

```
suraj-patel@SurajPatel:/$  
git config --global user.email "surajpatel9390@gmail.com"  
  
suraj-patel@SurajPatel:/$  
git config --global user.name "Suraj Patel"  
  
suraj-patel@SurajPatel:/$  
git config --list  
user.email=surajpatel9390@gmail.com  
user.name=Suraj Patel
```

## **2.3. Creating a repository**

For creating repository first you need to check the status if you get it like this

git status

```
④ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status  
fatal: not a git repository (or any of the parent directories): .git
```

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

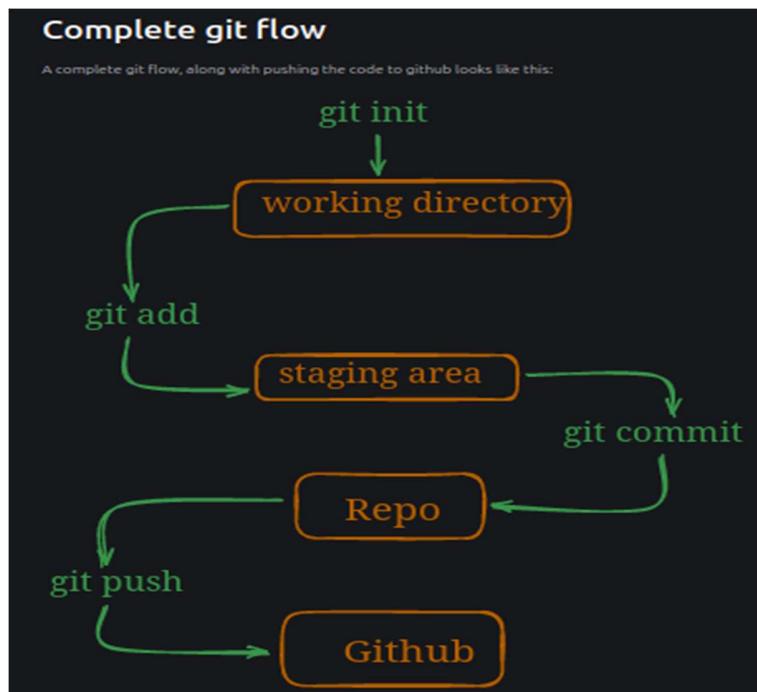
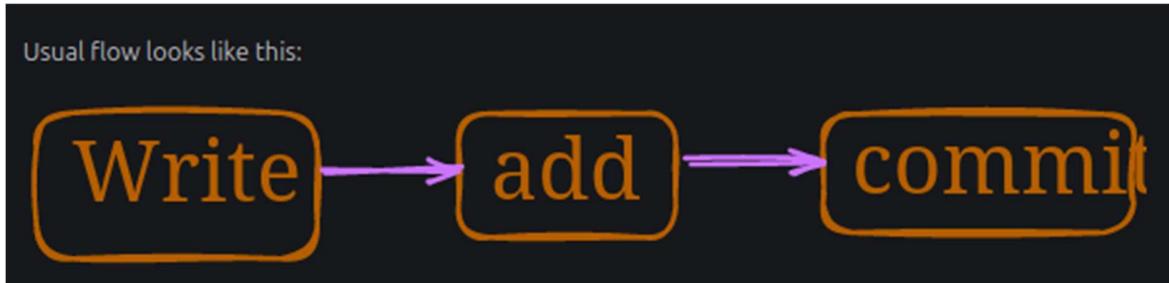
then creating repository

**git init is used to create repository**

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git init  
hint: Using 'master' as the name for the initial branch. This default branch name  
hint: is subject to change. To configure the initial branch name to use in all  
hint: of your new repositories, which will suppress this warning, call:  
hint:  
hint:   git config --global init.defaultBranch <name>  
hint:  
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and  
hint: 'development'. The just-created branch can be renamed via this command:  
hint:  
hint:   git branch -m <name>  
Initialized empty Git repository in /home/suraj-patel/Desktop/Git_Course/One/.git/
```

## 2.4. Commit Way

commit is a way to save your changes to your repository.



## 2.5. Stage

Stage is a way to tell git to track a particular file or folder. You can use the following command to stage a file:

```
• • •  
git init  
git add <file> <file2>  
git status
```

Here we are initializing the repository and adding a file to the repository. Then we can see that the file is now being tracked by git. Currently our files are in staging area, this means that we have not yet committed the changes but are ready to be committed.

Here git add <file1> <file2> to add is used to stage a file to commit it

Or other way

git add . To add all file

```
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git add .  
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
    new file:   file1.txt  
    new file:   file2.txt
```

And for remove a file you need to write code git rm –cached<file>...” to unstage

## 2.6. Commit

```
...  
git commit -m "commit message"  
git status
```

Here we are committing the changes to the repository. We can see that the changes are now committed to the repository. The **-m** flag is used to add a message to the commit. This message is a short description of the changes that were made. You can use this message to remember what the changes were. Missing the **-m** flag will result in an action that opens your default settings editor, which is usually VIM. We will change this to vscode in the next section.

Example

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git commit -m "first time commit"  
[master (root-commit) 5690072] first time commit  
 2 files changed, 2 insertions(+)  
  create mode 100644 file1.txt  
  create mode 100644 file2.txt
```

Now in the file1 we added something and now checking with status

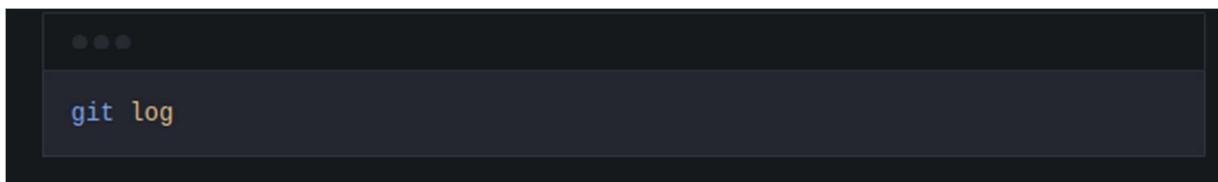
```
nothing to commit, working tree clean  
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
    (use "git restore <file>..." to discard changes in working directory)  
      modified:   file1.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

Now we commit file1 only to save in git

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git add file1.txt
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1.txt

● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git commit -m "something is added in file1"
[master aaf74c1] something is added in file1
 1 file changed, 2 insertions(+), 1 deletion(-)
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status
On branch master
nothing to commit, working tree clean
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

## 2.6. Logs



This command will show you the history of your repository. It will show you all the commits that were made to the repository. You can use the `--oneline` flag to show only the commit message. This will make the output more compact and easier to read.

### Example

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git log
commit aaf74c1606983dc7c13ed5d7eff3020d0f1b3dd4 (HEAD -> master)
Author: Suraj Patel <surajpatel9390@gmail.com>
Date:   Mon Jun 10 17:48:54 2024 +0530

    something is added in file1

commit 5690072b1997791503ebef2cf4e0234fce17e041
Author: Suraj Patel <surajpatel9390@gmail.com>
Date:   Mon Jun 10 16:36:13 2024 +0530

    first time commit
```

Now using `oneline` command to show log

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git log --oneline
aaf74c1 (HEAD -> master) something is added in file1
5690072 first time commit
```

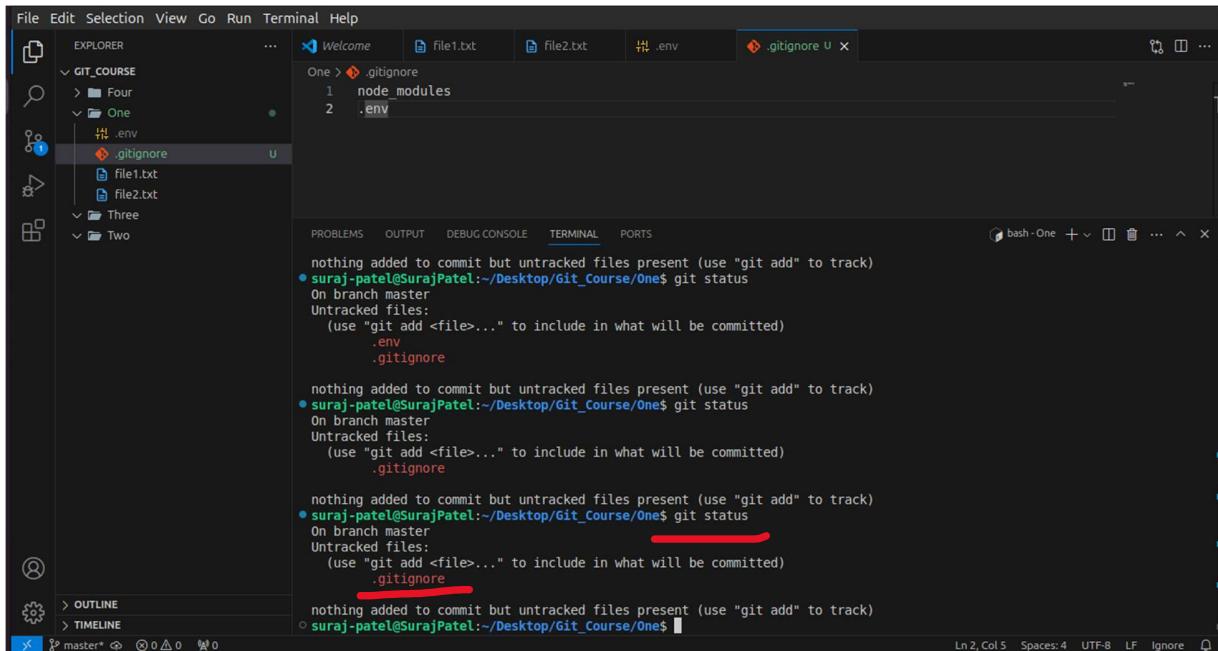
## 2.7. Gitignore

Gitignore is a file that tells git which files and folders to ignore. It is a way to prevent git from tracking certain files or folders. You can create a gitignore file and add list of files and folders to ignore by using the following command:

Example:

```
node_modules  
.env  
.vscode
```

Now, when you run the `git status` command, it will not show the `node_modules` and `.vscode` folders as being tracked by git.



The screenshot shows the VS Code interface. On the left, the Explorer sidebar displays a folder structure under 'GIT COURSE' with subfolders 'Four', 'One', 'Three', and 'Two'. Inside 'One', there are files '.env', '.gitignore', 'file1.txt', and 'file2.txt'. The '.gitignore' file is open in the center editor, containing the following content:

```
node_modules  
.env
```

In the bottom right terminal tab, the user runs the command `git status` three times. The first two runs show untracked files: '.env' and '.gitignore'. The third run shows no changes:

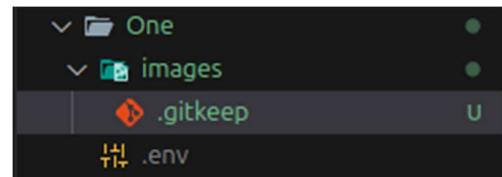
```
nothing added to commit but untracked files present (use "git add" to track)  
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status  
On branch master  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    .env  

```

git is ignored the .env file because of .gitignore file

# Blank Folder

By defaults blanks folder get ignore by git to keep blank folder we use .gitkeep



### 3. Branched in git

Branches are a way to work on different versions of a project at the same time. They allow you to create a separate line of development that can be worked on independently of the main branch. This can be useful when you want to make changes to a project without affecting the main branch or when you want to work on a new feature or bug fix.



Some developers can work on Header, some can work on Footer, some can work on Content, and some can work on Layout. This is a good example of how branches can be used in git.

#### **3.1. HEAD in git**

The HEAD is a pointer to the current branch that you are working on. It points to the latest commit in the current branch. When you create a new branch, it is automatically set as the HEAD of that branch.

Creating a new branch

```

git branch
git branch bug-fix
git switch bug-fix
git log
git switch master
git switch -c dark-mode
git checkout orange-mode

```

Some points to note:

- ❖ **git branch** (gives all branch names) - This command lists all the branches in the current repository. Example

EXPLORER    ...    File1.txt    File2.txt    .env    .gitignore    .gitkeep U    master X

GIT\_COURSE    One > .git > refs > heads > master  
1 ed6fec2d993bb69e515690aee20288644c3ea66d  
2

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git branch  
\* master  
suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$

- ❖ **git branch bug-fix**(Creating New Branch) - This command creates a new branch called bug-fix. Example:

EXPLORER    ...    File1.txt    File2.txt    .env    .gitignore    .gitkeep U    bug-fix X

GIT\_COURSE    One > .git > refs > heads > bug-fix  
1 ed6fec2d993bb69e515690aee20288644c3ea66d  
2

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

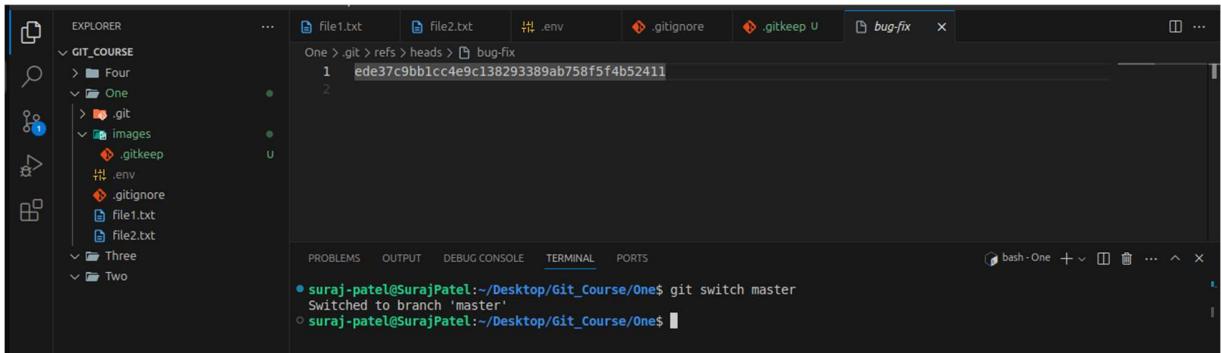
suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git branch  
\* master  
suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git branch bug-fix  
\* bug-fix  
suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$

But write now selected branch is master if you want to work on created branch you need to switch the branch

- ❖ **git switch bug-fix** (Switch the branch) - This command switches to the bug-fix branch. Example

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git branch
 * bug-fix
   master
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

- ❖ git log - This command shows the commit history for the current branch.
- ❖ git switch master (Switch to original path then what happened) - This command switches to the master branch.



BugFix file not showing because this file in another branch and branched name is (bug-fix)

- ❖ git switch -c dark-mode (create if exited and switch)- This command creates a new branch called dark-mode. the -c flag is used to create a new branch.

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git switch -c dark_mode
Switched to a new branch 'dark_mode'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git branch
  bug-fix
* dark_mode
  master
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

It is used to create branch and switch when branch is created and if branch is exited then it only switch

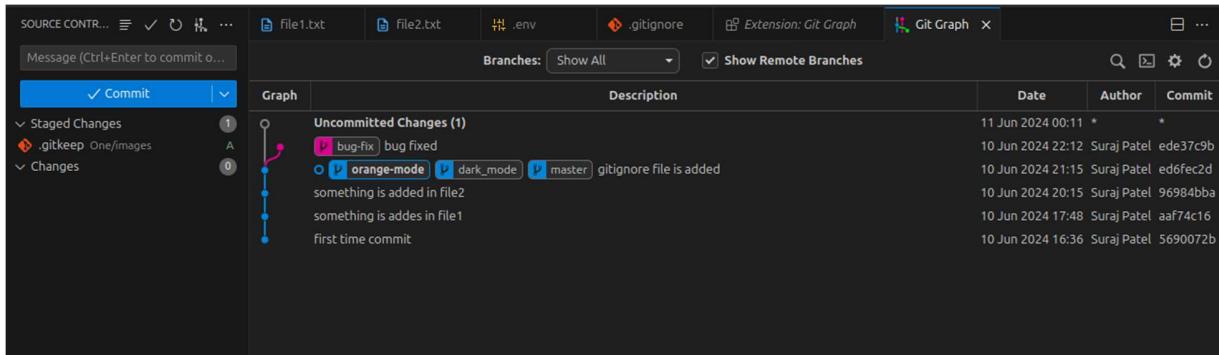
- ❖ git checkout orange-mode (It is used to switch and most used)- This command switches to the orange-mode branch.

```
master
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout orange-mode
error: pathspec 'orange-mode' did not match any file(s) known to git
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git branch orange-mode
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git branch
  bug-fix
  * dark_mode
    master
    orange-mode
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout orange-mode
Switched to branch 'orange-mode'
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ █
```

It is used to switch if exited and also tell that if branch is exited or not

- Commit before switching to a branch
- Go to .git folder and checkout to the HEAD file

## 3.2. Git Graph tool



## 3.3. Merging branches

### 3.3.1. Fast-forward merge

This one is easy as branch that you are trying to merge is usually ahead and there are no conflicts.

When you are done working on a branch, you can merge it back into the main branch. This is done using the following command:

```
git checkout main  
git merge bug-fix
```

```
graph LR; master(( )) --- master(( )); master --- merge(( )); bugFix(( )) --- bugFix(( )); bugFix --- merge;
```

Some points to note:

- `git checkout main` - This command switches to the `main` branch.
- `git merge bug-fix` - This command merges the `bug-fix` branch into the `main` branch.

This is a fast-forward merge. It means that the commits in the `bug-fix` branch are directly merged into the `main` branch. This can be useful when you want to merge a branch that has already been pushed to the remote repository.

## Example:

Git Graph Extension Screenshot

**Commits:**

Date	Author	Commit
11 Jun 2024 00:20	*	*
10 Jun 2024 22:12	Suraj Patel	ede37c9b
10 Jun 2024 21:15	Suraj Patel	ed6fec2d
10 Jun 2024 20:15	Suraj Patel	96984bba
10 Jun 2024 17:48	Suraj Patel	aaf74c16
10 Jun 2024 16:36	Suraj Patel	5690072b

**Terminal Output:**

```

master
orange-mode
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout orange-mode
Switched to branch 'orange-mode'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout bug-fix
A    images/.gitkeep
Switched to branch 'bug-fix'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

- In branch bug-fix you can see file BugFix but not in master branch(main) see below image

Git Graph Extension Screenshot

**Commits:**

Date	Author	Commit
11 Jun 2024 00:23	*	*
10 Jun 2024 22:12	Suraj Patel	ede37c9b
10 Jun 2024 21:15	Suraj Patel	ed6fec2d
10 Jun 2024 20:15	Suraj Patel	96984bba
10 Jun 2024 17:48	Suraj Patel	aaf74c16
10 Jun 2024 16:36	Suraj Patel	5690072b

**Terminal Output:**

```

Switched to branch 'orange-mode'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout bug-fix
A    images/.gitkeep
Switched to branch 'bug-fix'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout master
A    images/.gitkeep
Switched to branch 'master'
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

For merge file then BugFix file is also present in master

git merge bug-fix

EXPLORER

GIT COURSE

- > Four
- > One
  - > .git
  - > images
  - .env
  - .gitignore
  - BugFix.txt
  - file1.txt
  - file2.txt
- > Two
- > Three

BRANCHES: Show All  Show Remote Branches

Graph

	Description	Date	Author	Commit
Uncommitted Changes (1)				
master	bug fixed	11 Jun 2024 00:25	*	*
dark_mode	gitignore file is added	10 Jun 2024 22:12	Suraj Patel	ede37c9b
	something is added in file2	10 Jun 2024 21:15	Suraj Patel	ed6fec2d
	something is added in file1	10 Jun 2024 20:15	Suraj Patel	96984bba
	first time commit	10 Jun 2024 17:48	Suraj Patel	aaf74c16
		10 Jun 2024 16:36	Suraj Patel	5690072b

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

bash - One

```
Switched to branch 'master'
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git merge bug-fix
Updating ed6fec2..ede37c9
Fast-forward
 BugFix.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 BugFix.txt
• suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

OUTLINE TIMELINE

master+ 0 0 0 0 Git Graph

### 3.3.2. Not fast-forward merge



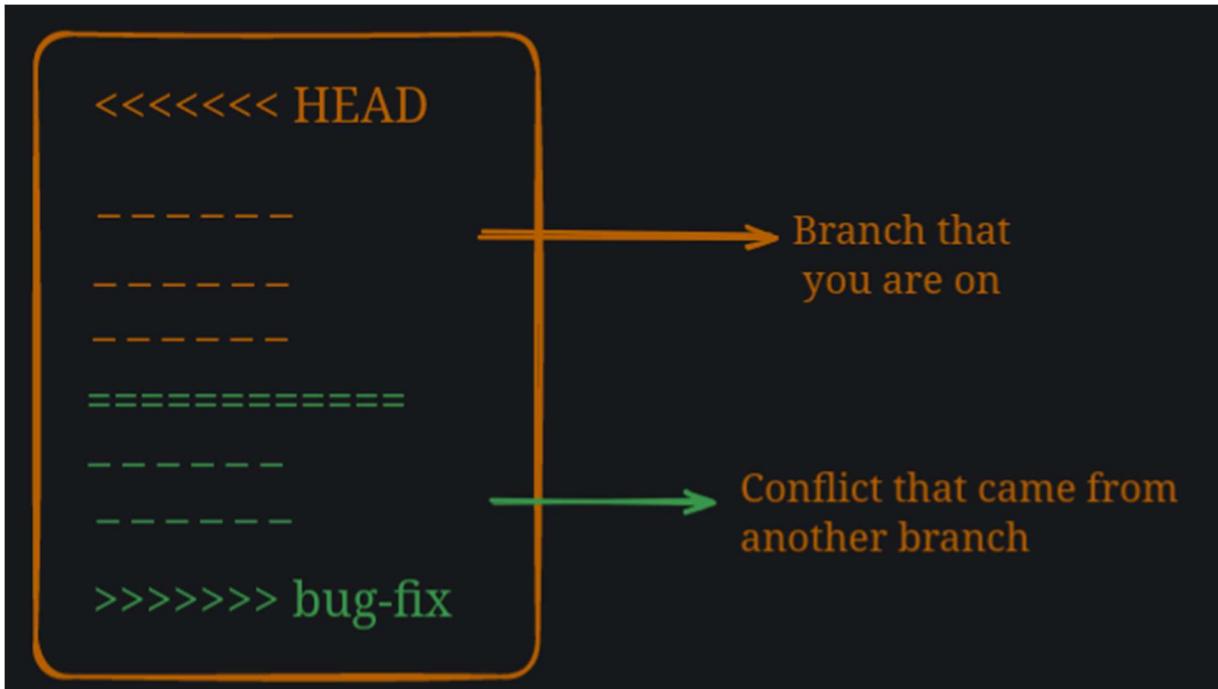
In this type of merge, the master branch also worked and have some commits that are not in the bug-fix branch. This is a not fast-forward merge.

When you are done working on a branch, you can merge it back into the main branch. This is done using the following command:

```
git checkout main  
git merge bug-fix
```

If the command are same, what is the difference between fast-forward and not fast-forward merge?

The difference is resolving the conflicts. In a fast-forward merge, there are no conflicts. But in a not fast-forward merge, there are conflicts, and there are no shortcuts to resolve them. You have to manually resolve the conflicts. Decide, what to keep and what to discard. VSCode has a built-in merge tool that can help you resolve the conflicts.

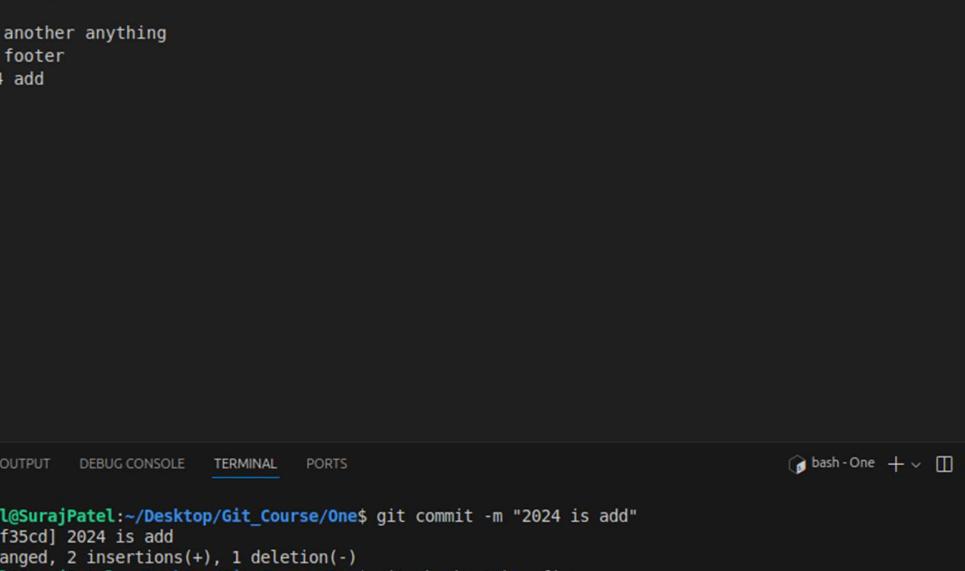


### 3.4. Managing conflicts

There is no magic button to resolve conflicts. You have to manually resolve the conflicts. Decide, what to keep and what to discard. VSCode has a built-in merge tool that can help you resolve the conflicts. I personally use VSCode merge tool. Github also has a merge tool that can help you resolve the conflicts but most of the time I handle them in VSCode and it gives me all the options to resolve the conflicts.

Overall it sounds scary to beginners but it is not, it's all about communication and understanding the code situation with your team members.

### **3.5. Conflicts creation and solving**



file1.txt File2.txt .env .gitignore Git Graph BugFix.txt

One > file2.txt

```
1 file 2
2 something is added in file2
3
4 add another anything
5 add footer
6 2024 add
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

bash - One

- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git commit -m "2024 is add"  
[master 14f35cd] 2024 is add  
1 file changed, 2 insertions(+), 1 deletion(-)
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout bug-fix  
Switched to branch 'bug-fix'
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout master  
Switched to branch 'master'

In the file2 there is so much content now switching from master(main) to bug-fix branch

The screenshot shows a terminal window with several tabs at the top: file1.txt, File2.txt (active), .env, .gitignore, Git Graph, and BugFix.txt. The main area displays the following command-line session:

```
One > file2.txt
1 file 2
2 something is added in file2
3

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash - One + × ☰ ... ^ ×
```

- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git merge bug-fix
Merge made by the 'ort' strategy.
BugFix.txt | 6 ++++++
images/.gitkeep | 0
2 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 images/.gitkeep
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git branch
 bug-fix
 dark-mode
\* master
 orange-mode
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git add .
suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git commit -m "2024 is add"
[master 14f35cd] 2024 is add
1 file changed, 2 insertions(+), 1 deletion(-)
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout bug-fix
Switched to branch 'bug-fix'
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout master
Switched to branch 'master'
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout bug-fix
Switched to branch 'bug-fix'
- suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$

Now we can see that in file2 there is no so much content and now bug-fix added footer also so there is conflict is created

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files: file1.txt, file2.txt (highlighted), Extension: Save Typing, .env, .gitignore, Git Graph, BugFix.txt.
- Terminal:** Displays the command history:
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git add .
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git commit -m "footer add in bug-fix file2"
  - [bug-fix 365dbd7] footer add in bug-fix file2
  - 1 file changed, 2 insertions(+)
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git status
  - On branch bug-fix
  - nothing to commit, working tree clean
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$

Footer is added not switching to master branch (main branch)

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files: file1.txt, file2.txt (highlighted), Extension: Save Typing, .env, .gitignore, Git Graph, BugFix.txt.
- Terminal:** Displays the command history:
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git add .
  - suraj-patel@SurajPatel:~/Desktop/Git Course/One\$ git commit -m "footer add in bug-fix file2"
  - [bug-fix 365dbd7] footer add in bug-fix file2
  - 1 file changed, 2 insertions(+)
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git status
  - On branch bug-fix
  - nothing to commit, working tree clean
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout master
  - error: pathspec 'mastr' did not match any file(s) known to git
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$ git checkout master
  - Switched to branch 'master'
  - suraj-patel@SurajPatel:~/Desktop/Git\_Course/One\$

Now you can see there is same file but content is different  
If you try merge with this code git merge bug-fix you get this type of window in vscode

```
file1.txt file2.txt Extension: Save Typing .env .gitignore Git Graph BugFix.I
```

One > file2.txt

1 file 2  
2 something is added in file2  
3  
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
4 <<<<< HEAD (Current Change)  
5 add another anything  
6 add footer  
7 2024 add  
8 =====  
9 2025 footer is added  
10 >>>>> bug-fix (Incoming Change)  
11

Resolve in Merge Editor

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout master
error: pathspec 'master' did not match any file(s) known to git
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git checkout master
Switched to branch 'master'
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git branch
  bug-fix
  dark_mode
* master
    orange-mode
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git merge big-fix
merge: big-fix - not something we can merge
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git merge bug-fix
Auto-merging file2.txt
CONFLICT (content): Merge conflict in file2.txt
Automatic merge failed; fix conflicts and then commit the result.
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

Now, you can select what you want to do from the following options.

File1.txt File2.txt ! .env .gitignore Git Graph BugFix.txt

One > file2.txt

1 file 2  
2 something is added in file2  
3  
4 add another anything  
5 add footer  
6 2024 add  
7  
8 header add  
9

Resolve in Merge Editor

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

bash - One + ×

```
Switched to branch 'master'  
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git merge bug-fix  
Auto-merging file2.txt  
CONFLICT (content): Merge conflict in file2.txt  
Automatic merge failed; fix conflicts and then commit the result.  
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status  
On branch master  
You have unmerged paths.  
(fix conflicts and run "git commit")  
(use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
 both modified: file2.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")  
suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

now you need to add file and commit

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git add file2.txt
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git commit -m "confilict resolved"
[master e11bf3c] conflict resolved
● suraj-patel@SurajPatel:~/Desktop/Git_Course/One$ git status
On branch master
nothing to commit, working tree clean
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/One$
```

You can also do manually like this

```
3 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare ...
4 <<<<< HEAD (Current Change)
5 add another marketing message section
6 add footer note
7 2024 is coming
8 =====
9 2025 footer note added
10 >>>>> bug-fix (Incoming Change)
11
```

Like this

```
one > file2.txt
1 this is my file two for git course.
2 adding more content to file two.
3 +
4 add another marketing message section
5 add another marketing message section
6 add footer note
7 2024 is coming
8
9
```

And now do same thing add and commit

## 3.6. Rename a branch

You can rename a branch using the following command:

```
...
git branch -m <old-branch-name> <new-branch-name>
```

## 3.7. Delete a branch

You can delete a branch using the following command:

```
git branch -d <branch-name>
```

## 3.8. Checkout a branch

You can checkout a branch using the following command:

```
git checkout <branch-name>
```

Checkout a branch means that you are going to work on that branch. You can checkout any branch you want.

## 3.9. List all branches

You can list all branches using the following command:

```
git branch
```

List all branches means that you are going to see all the branches in your repository.

## 4. diff stash and Tags

### 4.1. Git diff

The `git diff` is an informative command that shows the differences between two commits. It is used to compare the changes made in one commit with the changes made in another commit. Git consider the changed versions of same file as two different files. Then it gives names to these two files and shows the differences between them.

#### How to read the diff

- a -> file A and b -> file B
- ---- indicates the file A
- +++ indicates the file B
- @@ indicates the line number

Here the file A and file B are the same file but different versions.

Git will show you the changes made in the file A and file B. It will also show you the line number where the change occurred along with little preview of the change.

#### 4.1.1. Comparing Working Directory and Staging Area

```
...  
git diff
```

This command shows the unstaged changes in your working directory compared to the staging area. This command alone will not show you the changes made in the file A and file B, you need to provide options to show the changes.

#### **4.1.2. Comparing Staging Area with Repository**

```
git diff --staged
```

This command shows the changes between your last commit and the staging area (i.e., changes that are staged and ready to be committed).

#### **4.1.3. Comparing between branches**

```
git diff <branch-name-one> <branch-name-two>
```

This command compares the difference between two branches. For example:

Another way to compare the difference between two branches is to use the following command:

```
git diff branch-name-one..branch-name-two
```

#### **4.1.4. Comparing Specific Commits:**

```
git diff <commit-hash-one> <commit-hash-two>
```

This command compares the difference between two commits.

## 4.2. Git Stash

Stash is a way to save your changes in a temporary location. It is useful when you want to make changes to a file but don't want to commit them yet. You can then come back to the file later and apply the changes.

Conflicting changes will not allow you to switch branches without committing the changes. Another alternative is to use the `git stash` command to save your changes in a temporary location.

```
...  
git stash
```

This command saves your changes in a temporary location. It is like a stack of changes that you can access later.

Example:

If you added something in some file and you want to switch branch then you need to commit first but you want to not commit at that time then we use git stash for temporary save

### 4.2.1. Naming the stash

You can also name the stash by using the following command:

```
...  
git stash save "work in progress on X feature"
```

### 4.2.2. View the stash list

You can view the list of stashes by using the following command:

```
...  
git stash list
```

#### **4.2.3. Apply the stash**

You can apply the stash by using the following command:

```
...  
git stash apply
```

#### **4.2.4. Apply the specific stash**

You can apply the specific stash by using the following command:

```
...  
git stash apply stash@{0}
```

Here `stash@{0}` is the name of the stash. You can use the `git stash list` command to get the name of the stash.

#### **4.2.5. Applying and dropping the stash**

You can apply and drop the stash by using the following command:

```
...  
git stash pop
```

This command applies the stash and drops it from the stash list.

#### **4.2.6. Drop the stash**

You can drop the stash by using the following command:

```
...  
git stash drop
```

#### **4.2.7. Applying stash to a specific branch**

You can apply the stash to a specific branch by using the following command:

```
git stash apply stash@{0} <branch-name>
```

#### **4.2.8. Clearing the stash**

You can clear the stash by using the following command:

```
git stash clear
```

### **4.3. Git Tags**

Tags are a way to mark a specific point in your repository. They are useful when you want to remember a specific version of your code or when you want to refer to a specific commit. Tags are like sticky notes that you can attach to your commits.

#### **4.3.1. Creating a tag**

You can create a tag using the following command:

```
git tag <tag-name>
```

This command creates a new tag with the specified name. The tag will be attached to the current commit.

### **4.3.2. Create an annotated tag**

You can create an annotated tag using the following command:

```
git tag -a <tag-name> -m "Release 1.0"
```

This command creates an annotated tag with the specified name and message. The tag will be attached to the current commit.

### **4.3.3. List all tags**

You can list all tags using the following command:

```
git tag
```

This command lists all the tags in your repository.

### **4.3.4. Tagging a specific commit**

You can tag a specific commit using the following command:

```
git tag <tag-name> <commit-hash>
```

### **4.3.5. Push tags to remote repository**

You can push tags to a remote repository using the following command:

```
git push origin <tag-name>
```

#### **4.3.6. Delete a tag**

You can delete a tag using the following command:

```
...
```

```
git tag -d <tag-name>
```

#### **4.3.7. Delete tag on remote repository**

You can delete a tag on a remote repository using the following command:

```
...
```

```
git push origin :<tag-name>
```

## 5. Rebase and reflog

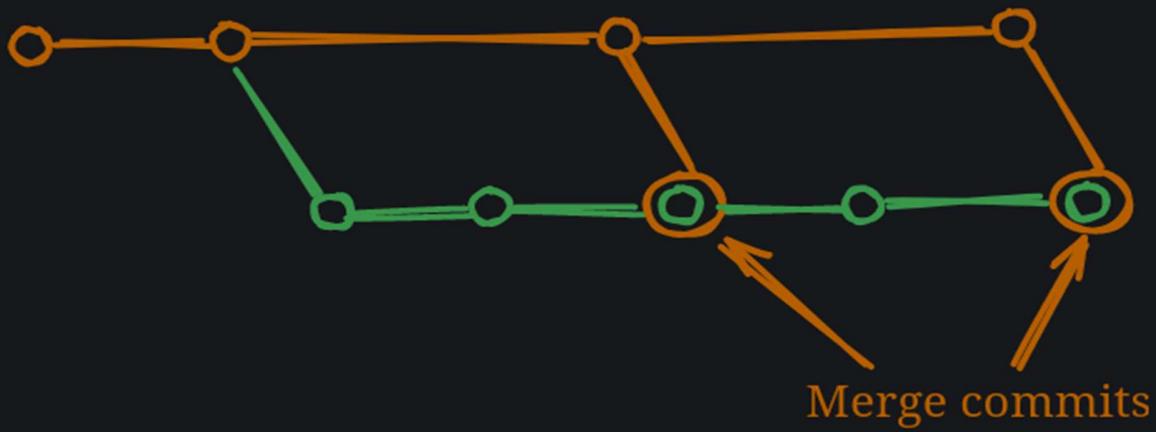
### 5.1. Rebase in git

Git rebase is a powerful Git feature used to change the base of a branch. It effectively allows you to move a branch to a new starting point, usually a different commit, by “replaying” the commits from the original base onto the new base. This can be useful for keeping a cleaner, linear project history.

Some people like to use rebase over the merge command because it allows you to keep the commit history cleaner and easier to understand. It also allows you to make changes to the code without affecting the original branch.

### 5.2. Merge commits

A merge commit is a commit that combines two or more commits into one. It is created when you merge two or more branches into a single branch. The merge commit contains all the changes from the original branches, and it is used to keep the project history clean and easy to understand.



### **5.3. Rebase**

## Rebase

Rebase is a powerful tool in Git that allows you to move a branch to a new starting point. It effectively replays the commits from the original base onto the new base. This can be useful for keeping a cleaner, linear project history.

Here's a flow example of using git rebase with all the commands involved:

Suppose you have a feature branch called `feature-branch` that you want to rebase onto the main branch.

### **5.4. Ensure you are on the branch you want to rebase:**

```
git checkout feature-branch  
git rebase main
```

This will replay the commits from `feature-branch` on top of the latest changes in `main`.

## **5.5. Resolve any conflicts:**

If there are any conflicts, you will need to resolve them manually. You can use the merge tool in VSCode to resolve the conflicts.

```
...  
git add <resolved-files>  
git rebase --continue
```

Try to avoid —force option when using rebase. It can cause issues with the project history. I have seen many horror stories of people using —force to fix conflicts.

## **5.6. Git reflog**

Git reflog is a command that shows you the history of your commits. It allows you to see the changes that you have made to your repository over time. This can be useful for debugging and understanding the history of your project.

## **5.7. View the reflog:**

```
...  
git reflog
```

This will show you the history of your commits. You can use the number at the end of each line to access the commit that you want to view.

## **5.8. Find specific commit**

You can find a specific commit using the following command:

```
...  
git reflog <commit-hash>
```

## **5.9. Recover lost commits or changes**

If you accidentally deleted a branch or made changes that are no longer visible in the commit history, you can often recover them using the reflog. First, find the reference to the commit where the branch or changes existed, and then reset your branch to that reference.

```
...  
git reflog <commit-hash>  
git reset --hard <commit-hash>
```

or you can use `HEAD@{n}` to reset to the nth commit before the one you want to reset to.

```
...  
git reflog <commit-hash>  
git reset --hard HEAD@{1}
```

## 6. Started with Github

### 6.1. Configure your config file

If you haven't done it already, you need to configure your git config file. You can do this by running the following command:

```
...  
git config --global user.email "your-email@example.com"  
git config --global user.name "Your Name"
```

This will set your email and name as your global settings. You can change these settings later by running the following command:

```
...  
git config --global user.email "your-email@example.com"  
git config --global user.name "Your Name"
```

Now you can check your config settings:

```
...  
git config --list
```

This will show you all the settings that you have changed.

### 6.2. Setup ssh key and add to github

If you haven't done it already, you need to setup ssh key and add it to your github account. You can do this by following the instructions on the Github website.

You can find the exact steps on the website for both Windows and MacOS. The steps are same for both, only apple users need to add the ssh key to their keychain.

### **6.2.1. Step 1: Generate a new SSH key**

To generate a new SSH key, open the terminal and run the following command:

```
...  
ssh-keygen -t ed25519 -C "your-email@chaicode.com"
```

Here ed25519 is the type of key that you are generating. This creates a new SSH key, using the provided email as label.

### **6.2.2. Save the key**

After generating the key, you need to save it to your computer. You can do this by running the following command:

| Enter a file in which to save the key (/Users/YOU/.ssh/id\_ALGORITHM): [Press enter]

At the prompt you can enter passphrase for the key or you can leave it blank. If you leave it blank, the key will be saved without a passphrase.

### **6.2.3. Add key to your ssh-agent**

After saving the key, you need to add it to your ssh-agent. You can do this by running the following command:

Here it is best to refer above link for more information, as Github has a lot of information on this. There is no point in repeating it here.

### **6.2.4. Add key to github**

Use the webui to add the key to your github account. You can do this by following the instructions on the [Github website](#).

## **6.3. Adding code to remote repository**

Now that you have setup your ssh key and added it to your github account, you can start pushing your code to the remote repository.

Create a new Repo on your system first, add some code and commit it.

```
git init  
git add <files>  
git commit -m "commit message"
```

### **6.3.1. Check remote url setting**

You can check the remote url setting by running the following command:

```
git remote -v
```

This will show you the remote url of your repository.

Example after adding remote repository checking and we get this

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/Three$ git remote add origin https://github.com/SurajPatel04/Try.git  
● suraj-patel@SurajPatel:~/Desktop/Git_Course/Three$ git remote -v  
origin https://github.com/SurajPatel04/Try.git (fetch)  
origin https://github.com/SurajPatel04/Try.git (push)  
○ suraj-patel@SurajPatel:~/Desktop/Git_Course/Three$
```

Ln 1, Col 11 Spaces: 4 UTF-8 L

### **6.3.2. Add remote repository**

You can add a remote repository by running the following command:

```
| git remote add origin <remote-url>
```

Here `<remote-url>` is the url of the remote repository that you want to add and origin is the name of the remote repository. This origin is used to refer to the remote repository in the future.

```
● suraj-patel@SurajPatel:~/Desktop/Git_Course/Three$ git remote add origin https://github.com/hiteshchoudhary/chai-something.git
```

### **6.3.3. Push code to remote repository**

```
| git push remote-name branch-name
```

Here `remote-name` is the name of the remote repository that you want to push to and `branch-name` is the name of the branch that you want to push.

```
...  
git push origin main
```

Or use `git push -u origin main`

### **6.3.4. Setup an upstream remote**

Setting up an upstream remote is useful when you want to keep your local repository up to date with the remote repository. It allows you to fetch and merge changes from the remote repository into your local repository.

To set up an upstream remote, you can use the following command:

```
...  
git remote add upstream <remote-url>
```

or you can use shorthand:

```
...  
git remote add -u <remote-url>
```

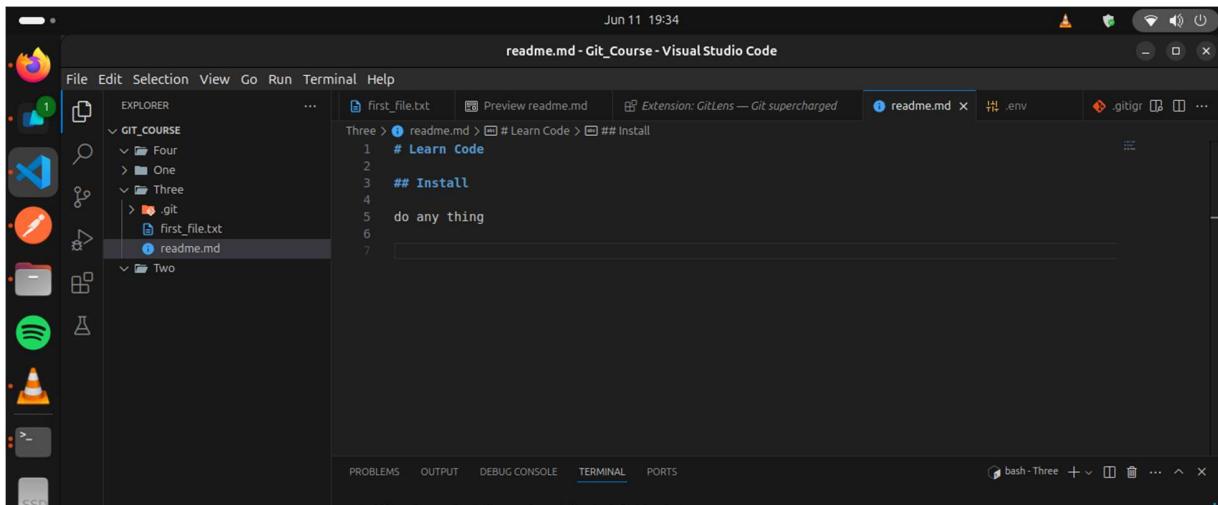
You can do this at the time of pushing your code to the remote repository.

```
...  
git push -u origin main
```

This will set up an upstream remote and push your code to the remote repository.

This will allow you to run future commands like `git pull` and `git push` without specifying the remote name.

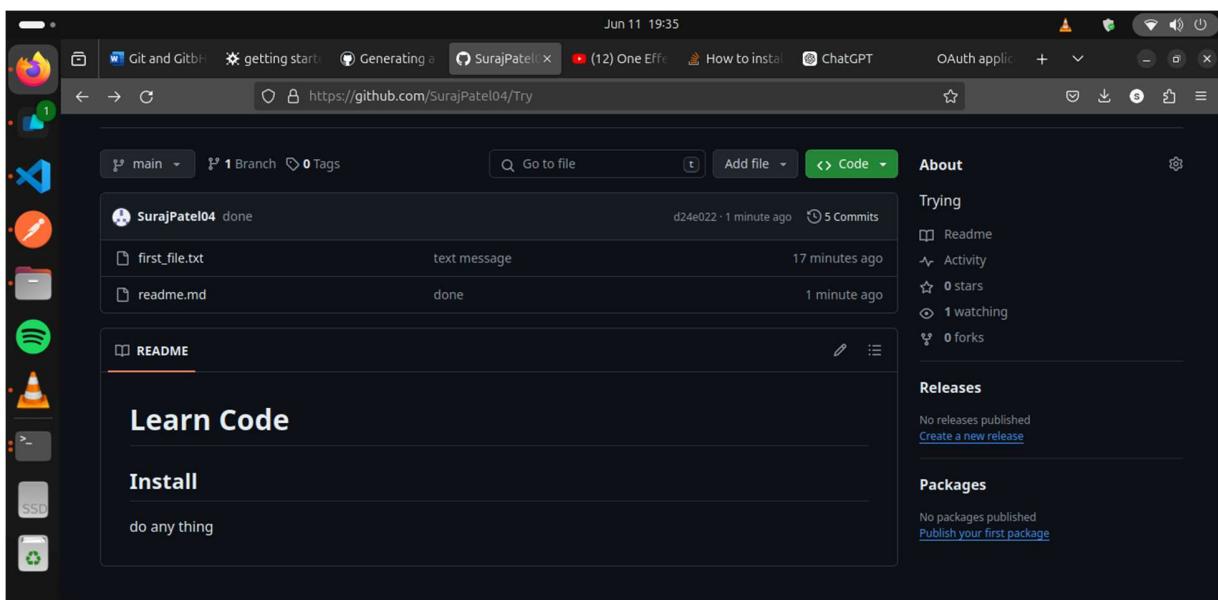
## 6.4. Readme.md file



The screenshot shows the Visual Studio Code interface with a dark theme. The Explorer sidebar on the left shows a folder structure under 'GIT\_COURSE' with subfolders 'One', 'Two', and 'Three'. Inside 'Three', there are files 'first\_file.txt' and 'readme.md'. The main editor area displays the contents of 'readme.md':

```
Jun 11 19:34
readme.md - Git_Course - Visual Studio Code
File Edit Selection View Go Run Terminal Help
readme.md - Git_Course - Visual Studio Code
readme.md
Extension: GitLens — Git supercharged
readme.md
.env
.gitignore
...
readme.md
# Learn Code
## Install
do any thing
```

After push you get this

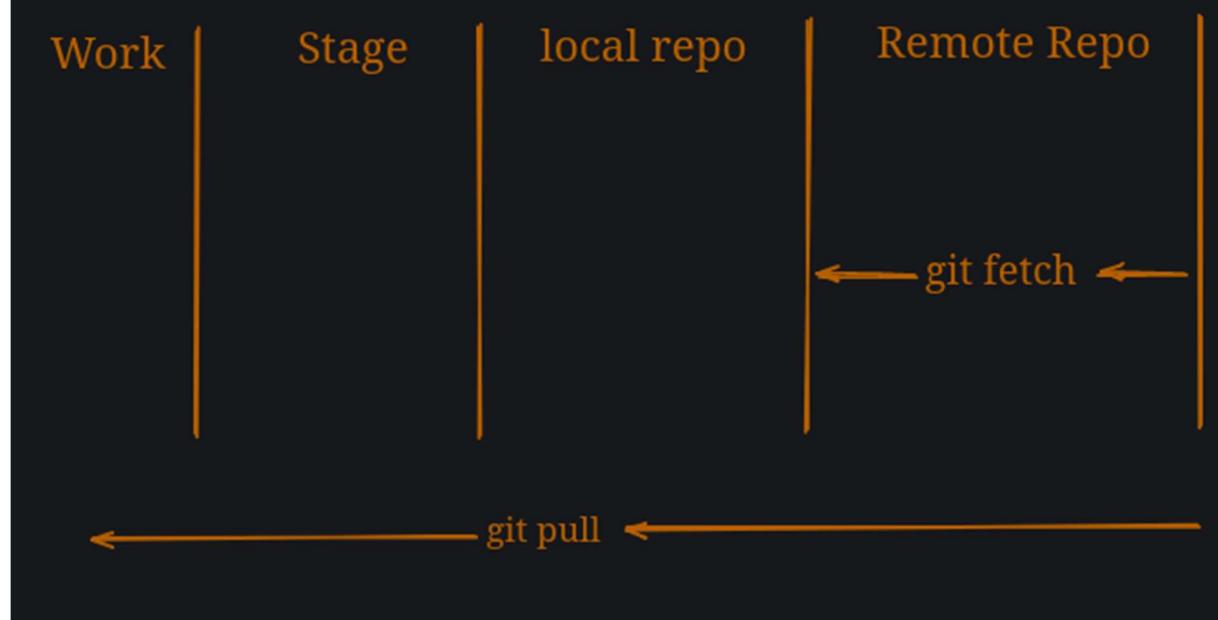


## **6.5. Get code from remote repository**

There are two ways to get code from a remote repository:

- fetch the code
- pull the code

Fetch the code means that you are going to download the code from the remote repository to your local repository. Pull the code means that you are going to download the code from the remote repository and merge it with your local repository.



### **6.5.1. Fetch code**

To fetch code from a remote repository, you can use the following command:

```
...  
git fetch <remote-name>
```

Here `<remote-name>` is the name of the remote repository that you want to fetch from.

## 6.5.2. Pull code

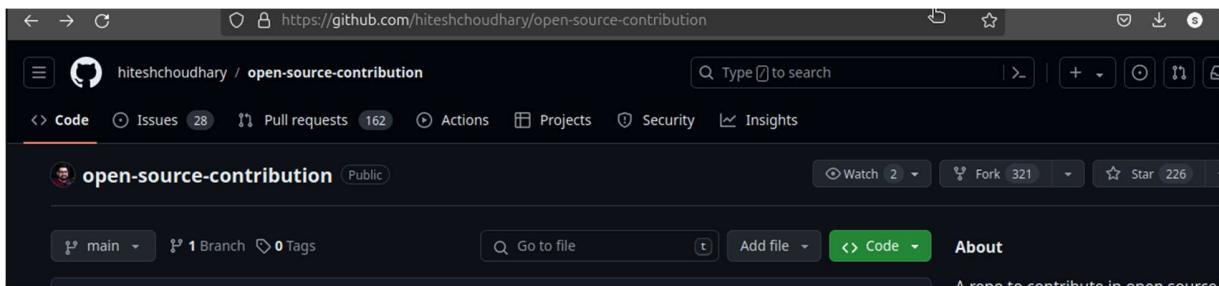
To pull code from a remote repository, you can use the following command:

```
...  
# git pull <remote-name> <branch-name>  
git pull origin main
```

Here `<remote-name>` is the name of the remote repository that you want to pull from and `<branch-name>` is the name of the branch that you want to pull.

# 7. Open Source

First step to create Fork



The clone by this step

Copy

HTTPS

and go to vs code and type `git clone <url>`

Then to you work and then add commit and push

If you want to send the changes then click on contribution button

