**GIT AND GITHUB**

**1. Git vs GitHub: Asli Difference**

Log aksar confuse hote hain, par inka rishta **"Mobile Phone"** aur **"Instagram"** jaisa hai.

* **Git (The Tool):** Yeh ek software hai jo aapke computer par chalta hai. Iska kaam hai "Version Control". Matlab, aapki files mein jo bhi changes hote hain, yeh unka hisaab rakhta hai.
* **GitHub (The Platform):** Yeh ek website hai jahan aap apna Git-tracked code internet par save karte ho.

**Example:** Socho aap ek essay likh rahe ho. **Git** woh "Save" button aur "History" hai jo track karti hai ki aapne kal kya likha tha. **GitHub** woh "Google Drive" ya "Cloud" hai jahan aap us file ko upload kar dete ho taaki aapka dost bhi use dekh sake.

**2. Version Control System (VCS) kya hai?**

Aapne "Game Checkpoint" wali baat bilkul sahi pakdi!

Jab hum code likhte hain, toh aksar kuch naya try karte hain aur pura code phat (crash) jata hai. Agar aapke paas **Version Control** nahi hai, toh aap puraana wala sahi code kho doge.

* **Snapshots:** Git har badlav ko ek "Snapshot" (Photo) ki tarah save karta hai.
* **Collaboration:** Jab 10 log ek hi project par kaam kar rahe hon, toh VCS track karta hai ki kisne kaunsi line change ki, taaki koi kisi ka kaam delete na kar de.

**3. History: SCCS se Git tak**

Pehle ke systems (jaise SCCS, SVN) thode slow aur rigid hote the.

* **Centralized VCS (Old):** Isme history ek hi server par hoti thi. Agar server down, toh aapka kaam ruk gaya.
* **Distributed VCS (Git):** Git "Distributed" hai. Iska matlab hai ki har developer ke computer par project ki **puri history** ki ek copy hoti hai. Internet nahi bhi hai, tab bhi aap "Commit" (save) kar sakte ho.

**4. SSH Key Setup: Password ka Zamana Gaya!**

Aapne notice kiya hoga ki ab GitHub password nahi mangta jab aap code bhejte (push karte) ho.

* **SSH Key:** Yeh ek "Digital Signature" jaisa hai.
* Aap apne computer par ek **Public Key** aur ek **Private Key** generate karte ho.
* Public Key aap GitHub ko de dete ho. Ab jab bhi aapka computer GitHub se baat karega, GitHub check karega ki "Haan, yeh wahi machine hai jiske paas matching key hai."
* Yeh security ke liye bahut zaroori hai.

**5. Learning Path: Steps to Success**

Aapne jo steps bataye hain, wahi best tarika hai:

1. **Basics:** Git install karna aur git init, git add, git commit seekhna.
2. **Daily Use:** Roj ke changes ko save karna.
3. **Face Problems:** Jab "Merge Conflicts" aate hain (do logo ne ek hi line change kar di), tab asli learning hoti hai.
4. **GitHub:** Phir apne code ko cloud par bhej kar duniya ko dikhana.

**Key Terms Jo Aapko Yaad Rakhne Hain:**

* **Commit:** Ek checkpoint save karna.
* **Push:** Local code ko GitHub par bhejna.
* **Pull:** GitHub se naya code apne computer par lana.
* **Branch:** Project ki ek alag "copy" banakar usme naya feature try karna (binna main code ko chede).

**Version Control** ka matlab hai apne kaam ki ek **"Time Machine"** taiyar karna.

Asal mein, jab hum kisi project par kaam karte hain, toh hum files mein bar-baar badlav (changes) karte hain. Version Control un saare badlavon ka hisaab rakhta hai taaki aap kabhi bhi purane version par wapas ja sakein.

Isko teen bade points mein detail mein samajhte hain:

**1. History Tracking (The "What happened?" part)**

Socho aap ek college assignment likh rahe ho. Aapne file save ki assignment\_v1.docx. Phir kuch change kiya aur save kiya assignment\_final.docx, phir assignment\_final\_final\_v2.docx. Ye bahut messy ho jata hai.

**Version Control** is jhanjhat ko khatam karta hai. File ka naam ek hi rehta hai, lekin system (Git) background mein har save (commit) ka ek snapshot le leta hai.

* Aap dekh sakte ho ki 2 din pehle code kaisa dikhta tha.
* Aap check kar sakte ho ki kaunsi line kab aur kyun badli gayi.

**2. Collaboration (Team Work)**

Jab ek hi project par 5-6 developers kaam karte hain, toh sabse bada darr hota hai ki **"Kahin mera code kisi aur ke code ko overwrite na kar de."**

Version Control sabko apni-apni copy par kaam karne deta hai aur baad mein smart tarike se unhe **Merge** (jodna) karta hai. Agar do logo ne ek hi line change ki hai, toh Git aapko bata dega (jise hum *Merge Conflict* kehte hain) aur aapse puchega ki kaunsa wala rakhna hai.

**3. Experimentation (Branching)**

Maan lo aapka project ekdum sahi chal raha hai (Stable version). Ab aap usme ek naya aur risky feature try karna chahte ho.

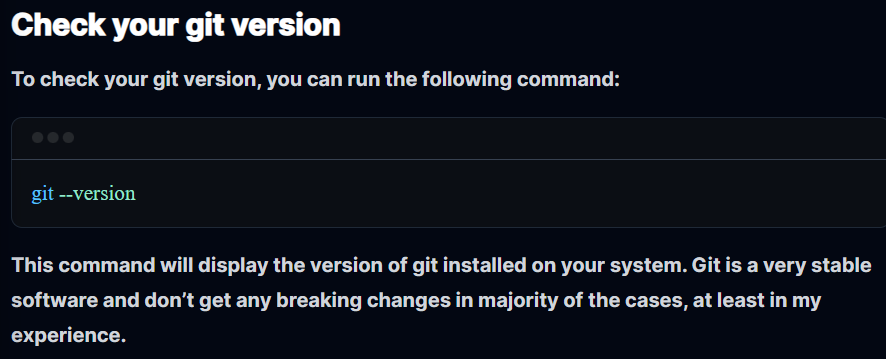
* Version Control mein aap ek **Branch** banate ho.
* Ye branch ek "Alternative Reality" jaisi hai. Aap yahan kitni bhi tod-fod karo, main code (Master/Main branch) safe rahega.
* Agar experiment successful raha, toh use main code mein jod do. Agar fail hua, toh us branch ko delete kar do—main project par koi asar nahi padega.

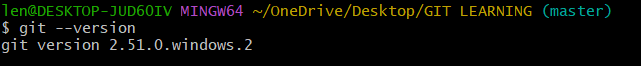
**Repository & Branch (Terminology)**

Jaise aapne kaha, Git ki apni ek duniya hai.

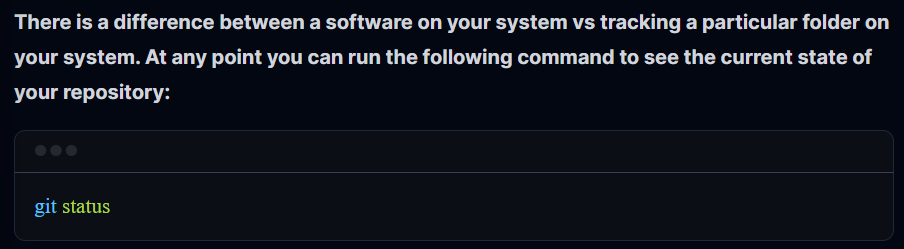
* **Repository (Repo):** Project ka "Home". Ye sirf ek folder nahi hai, ye ek *Smart Folder* hai jise pata hai ki kal aapne kya code likha tha.
* **Branch:** Bilkul sahi, ye **"Alternative Timeline"** hi hai. Maan lo aap ek game khel rahe ho aur ek "What if" scenario try karna chahte ho bina main game kharab kiye—woh Branch hai.

**GIT VERSION**





**GIT STATUS**



**git status**: Ye aapka sabse bada dost hai. Har 2 minute mein ise chalao. Ye batata hai: "Kaunsi file change hui? Kya add karna bhool gaye?"

**git status**

Ye command "Inspector" hai. Ise baar-baar chalao. Ye aapko batayega:

* Kaunsi files modify hui hain?
* Kya aapne git add kiya?
* Kya kuch commit karna baaki hai?

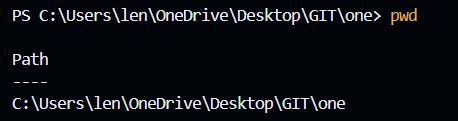


**GO INSIDE A FOLDER OR GET OUT OF IT USING TERMINAL**



**IMPORTANT**





**GIT CONFIG**

**Configuration: Git ko apna naam-pata batana**

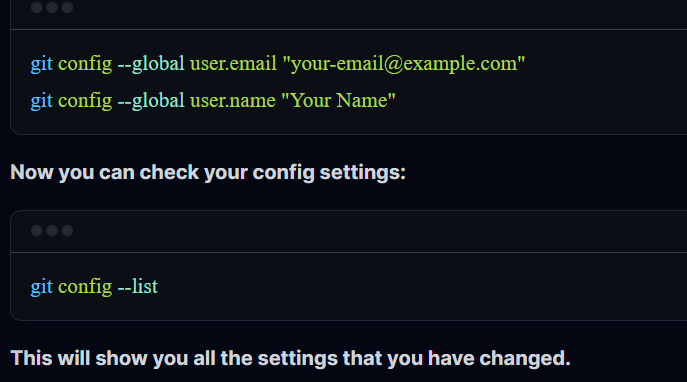
Jab aap pehli baar Git install karte ho, toh use batana padta hai ki "Main kaun hoon?".

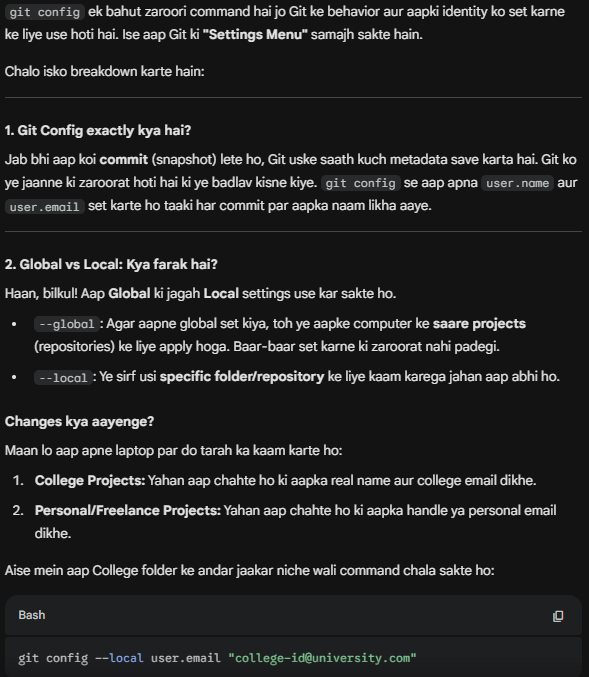
Bash

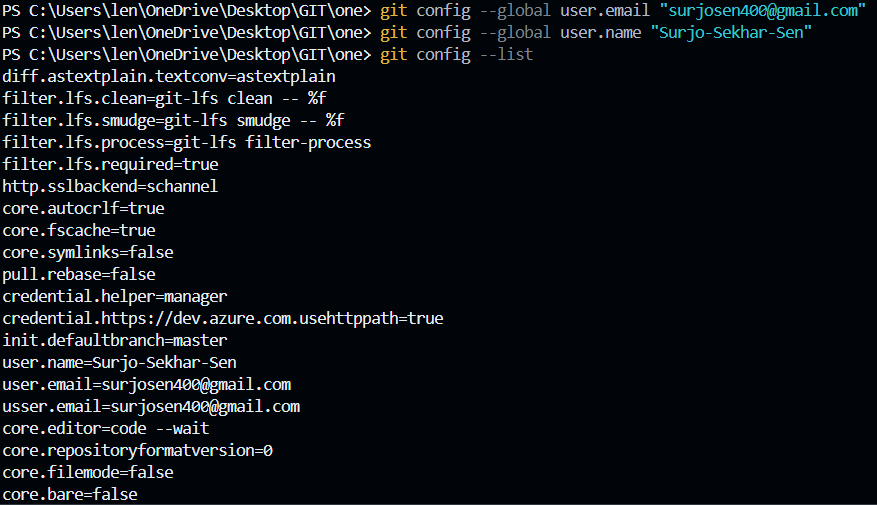
git config --global user.name "Aapka Naam"

git config --global user.email "aapka@email.com"

* **Kyun?** Kyunki jab aap koi change (commit) karoge, toh Git us par aapka thappa (stamp) laga dega. Jab 10 log ek project par kaam karte hain, toh isi se pata chalta hai ki kisne "code phada" aur kisne "fix kiya".

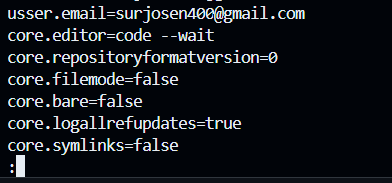






**IMPORTANT**

**TO GET OUT OF THIS SITUATION**



Vim se nikalne ke liye :wq type karna padta hai !!

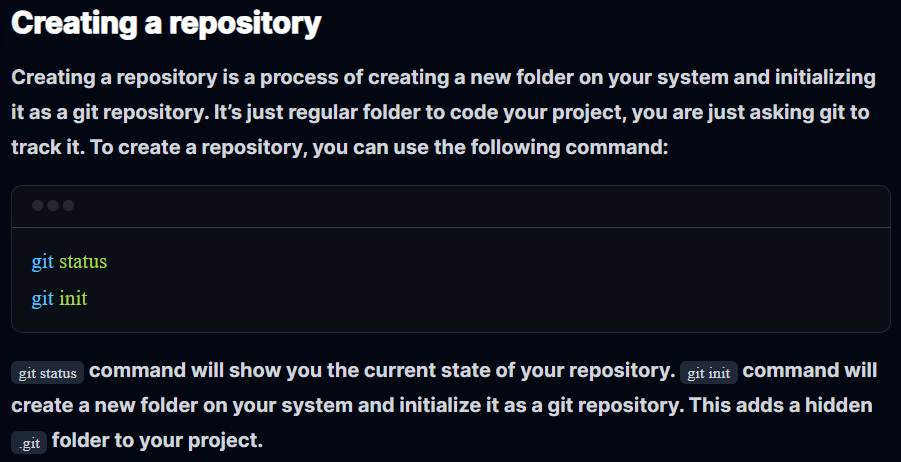
**GIT INIT**

**git init: Naye project ki shuruat. Isse .git folder banta hai jo sab kuch track karta hai.**

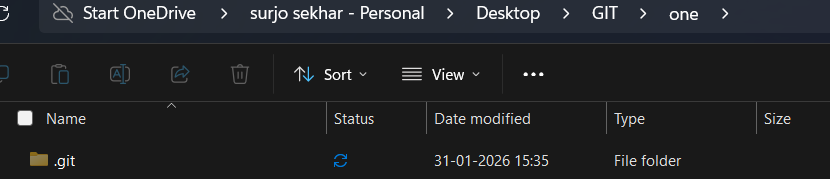
**Repository: Normal Folder vs Git Repo**

Aapne sahi kaha tha, Repo ek folder hi hai, par usme **.git** naam ka ek hidden folder hota hai.

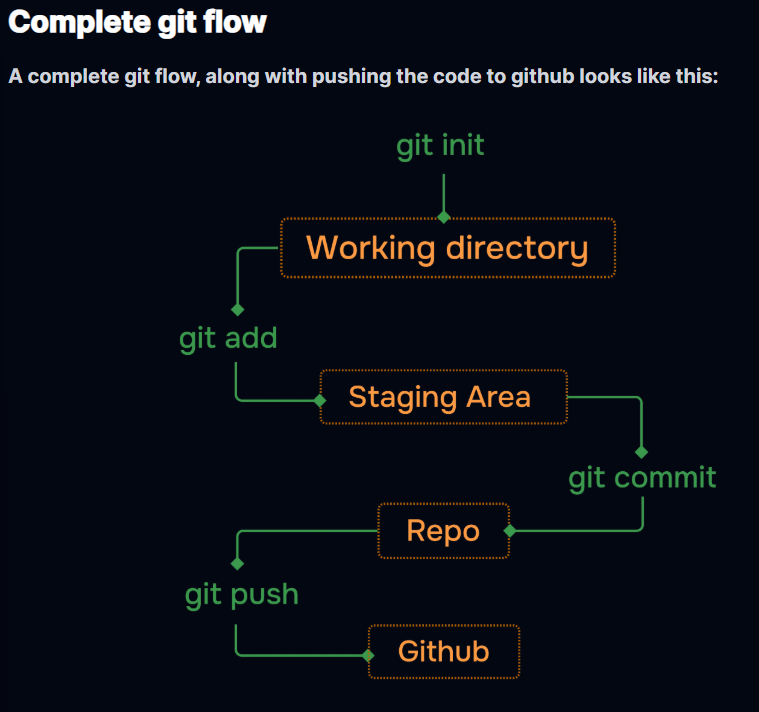
* **Normal Folder:** Sirf files rakhta hai.
* **Git Repo (git init ke baad):** Ye file ki **atma (soul)** ko track karta hai. Agar aapne 10 din pehle ek line delete ki thi, toh Git ke paas uska record hai.







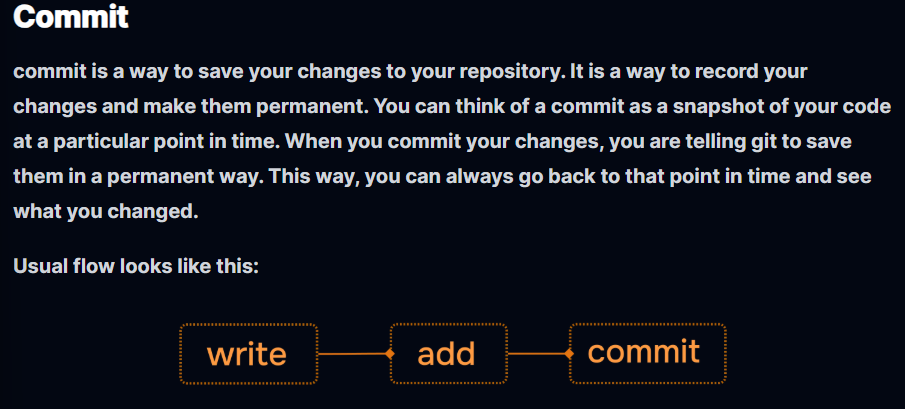
**GIT FLOW**



Git mein code "Save" karne ke teen steps hote hain. Ise **Git Workflow** kehte hain:

1. **Working Directory (The Kitchen):** Yahan aap kaam kar rahe ho. Files create kar rahe ho, code likh rahe ho. Git ko pata hai ki aapne change kiya hai, par usne abhi tak kuch "save" nahi kiya. (Yahan files **Red** dikhti hain git status mein).
2. **Staging Area (The Packing Area):** Jab aap **git add filename** karte ho, toh file Staging Area mein chali jati hai. Iska matlab hai: *"Maine ye file ready kar di hai, agle snapshot mein ise shamil kar lena."* (Yahan files **Green** dikhti hain).
3. **Local Repository (The Vault):** Jab aap **git commit -m "message"** karte ho, toh Git ek snapshot leta hai aur use hamesha ke liye apne database (The Vault) mein save kar deta hai.

**GIT COMMIT**

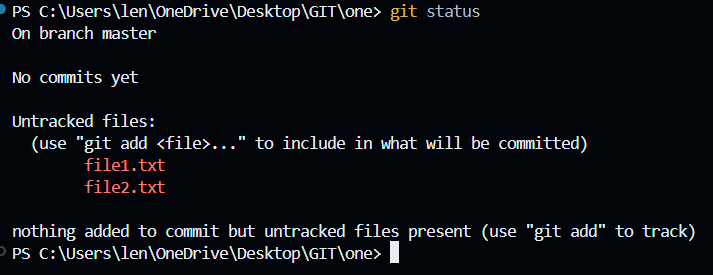


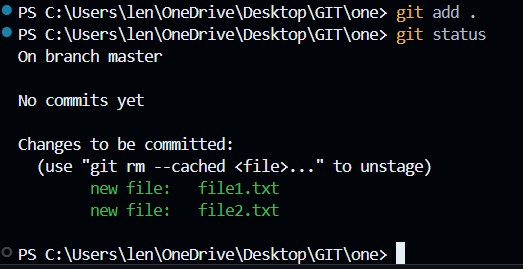
git add . Saari files ko Staging Area (packing area) mein bhejta hai.

git commit -m "your msg" Snapshot lekar hamesha ke liye save kar deta hai.

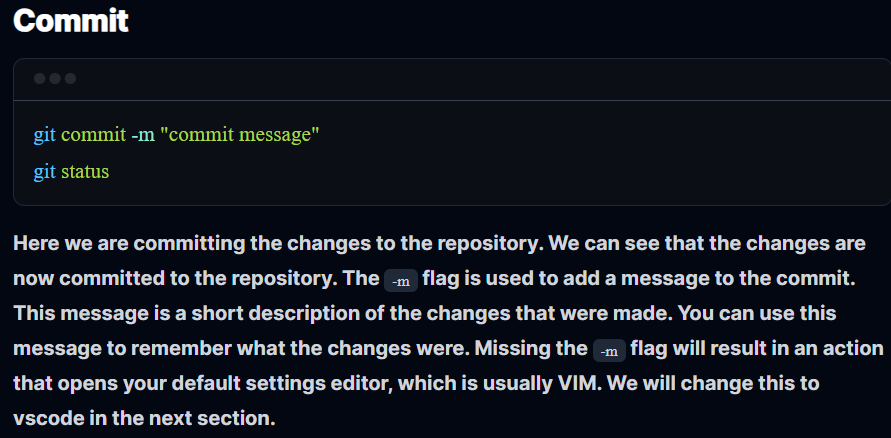


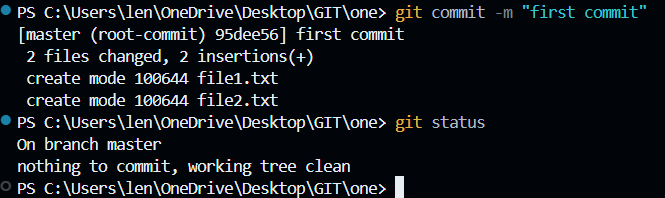
**WITHOUT ADD IF WE CHECK STATUS**





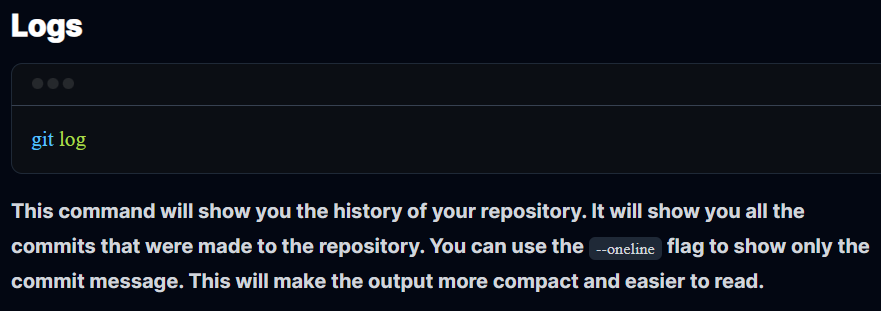
**GIT COMMIT**

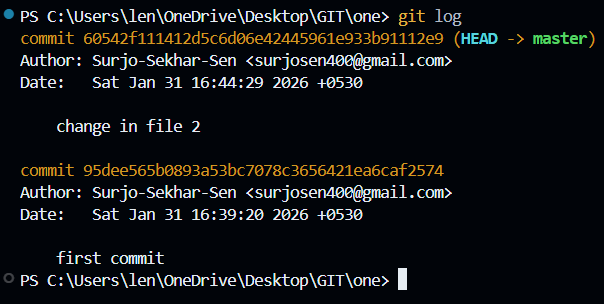


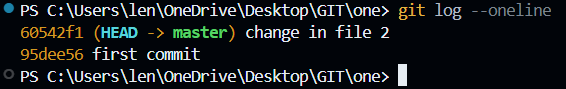


**GIT LOG**

**git log** : Ye aapke project ki "Timeline" hai. --oneline se history short aur saaf dikhti hai.





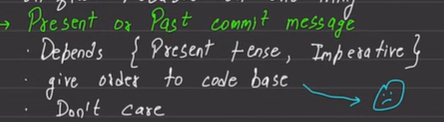


**IMPORTANT**

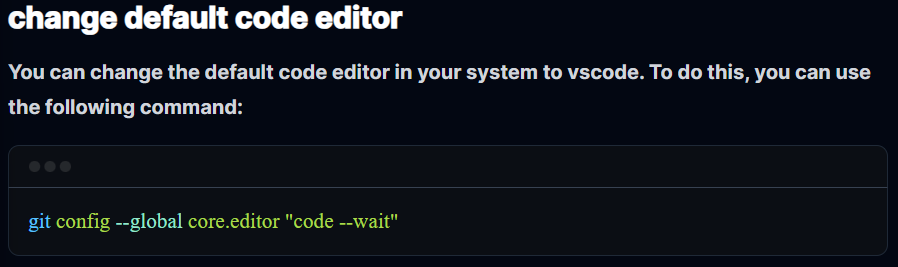
**ATOMIC COMMITS**

**Atomic Commits:** Iska matlab hai ki ek commit mein sirf ek hi "kaam" hona chahiye. Agar aapne 'Login Page' banaya aur 'Header' bhi fix kiya, toh dono ke alag-alag commit karo. Isse agar baad mein 'Login' mein bug aaya, toh aap sirf wahi part rollback kar sakte ho.

**HOW TO WRITE COMMIT MESSAGES PROFESSIONALY**



**Editor Change (VS Code Setup)**

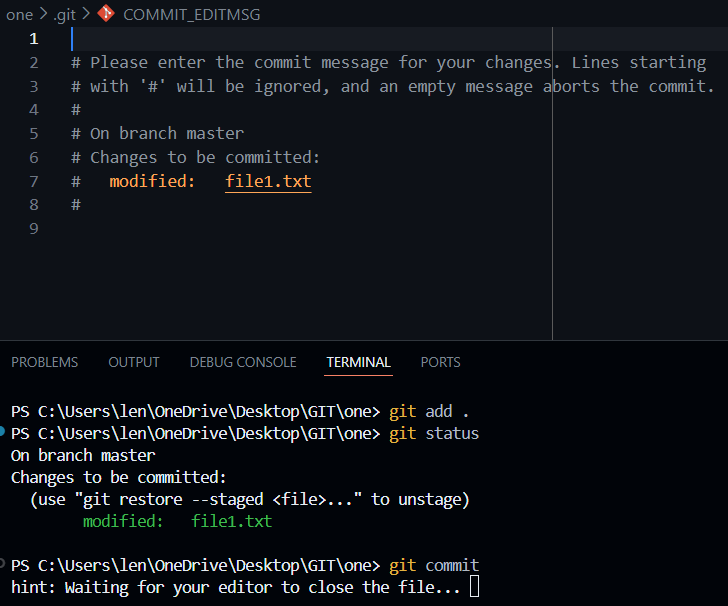


By default, Git ek purana editor use karta hai jise **Vim** kehte hain. Vim se bahar nikalna naye logo ke liye mushkil hota hai. Isliye hum ye command chalate hain:

Bash

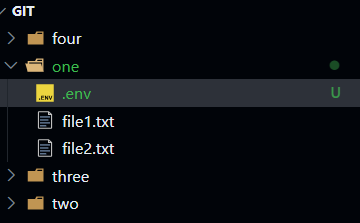
git config --global core.editor "code --wait"

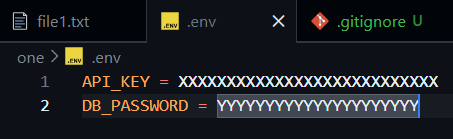
Ab jab bhi Git ko aapse koi bada message likhwana hoga, woh apne aap **VS Code** khol dega.

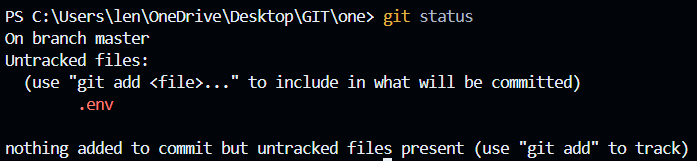


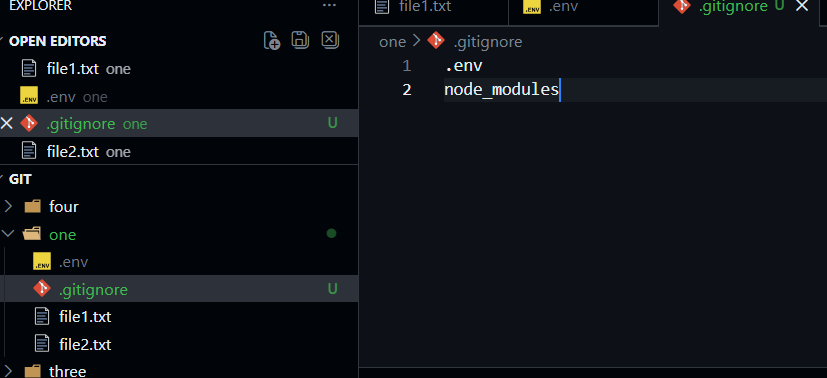
**GIT IGNORE AND GIT KEEP**

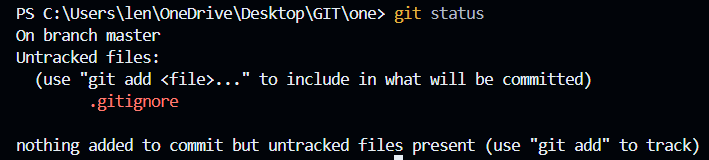
**.gitignore**: Kuch files aisi hoti hain jo hum GitHub pe nahi bhejna chahte (jaise passwords wali .env file, ya heavy folders jaise node\_modules). Unka naam is file mein likh do, Git unhe ignore kar dega.



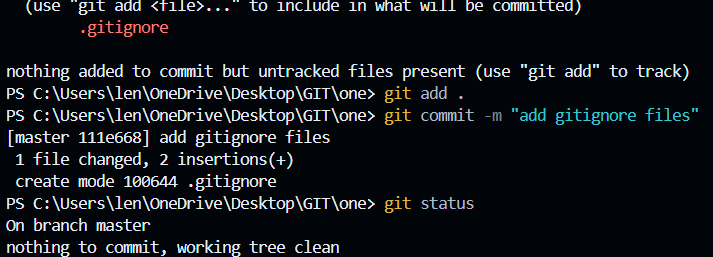




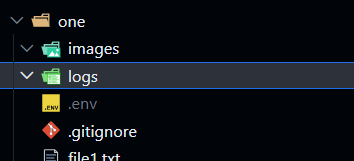




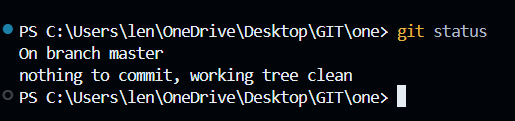
Here we see .env file isn’t coming as untracked files. It is because we put that file in gitignore.



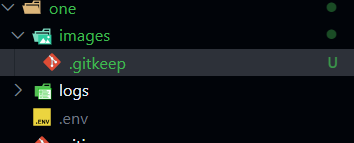
**GIT KEEP**



Here logs and images are two folders jo banaya gaya hai one folder ke andar. Ye dono hi folders empty hai. Agar ham inn empty folders ko le aur git status check kare to kuch bhi nahi aayega..



Aisa isliye kyuki dono folders empty hai. Agar ham kuch folders empty rakhna chahte hai but repository me ham unko lana chahte hai to usko lane ke liye hame gitkeep ka use karna padega..



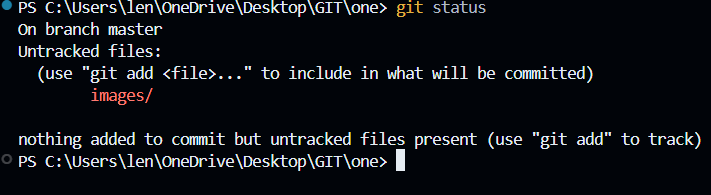


Abb bass add aur commit karlo!!

**BRANCHING**

**Branches: Alternative Timelines**

Branch ka matlab hai project ki ek copy banana jahan aap bina main code (main branch) ko chede kuch bhi naya try kar sako.

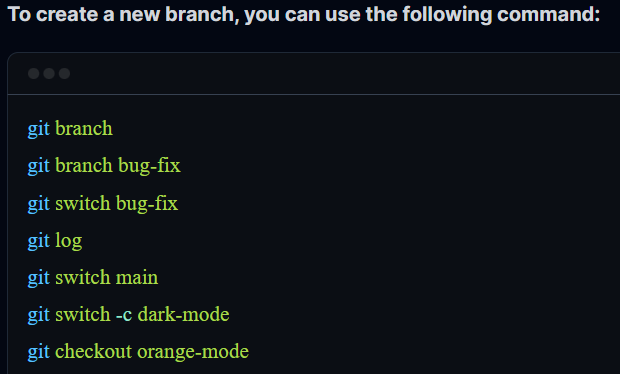
* **Header, Footer, Content wala example:** Socho 4 dost ek website bana rahe hain. Sab log agar ek hi file mein likhenge toh hamesha "Conflict" aayega.
* Isliye har koi apni alag branch banata hai (feature-header, feature-footer). Jab kaam khatam ho jata hai, tab sabko **Main** mein merge kar dete hain.



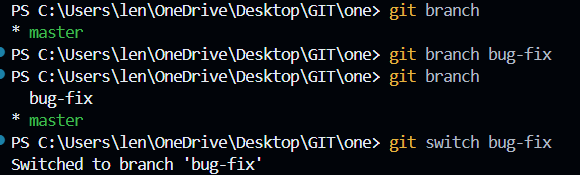
**HEAD: "You are here" ka pointer**

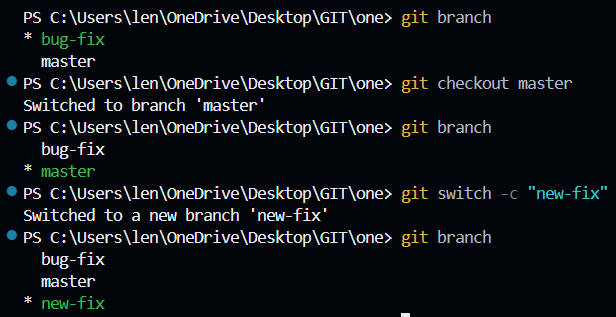
Git mein **HEAD** ek pointer (ungli) ki tarah hai. Ye batata hai ki aap abhi kis commit ya branch par khade ho.

* Jab aap git switch bug-fix karte ho, toh HEAD 'main' se hatkar 'bug-fix' par chala jata hai.
* **Master vs Main:** Pehle default naam 'master' hota tha, ab 'main' hai. Dono mein koi technical farak nahi hai, sirf naam ka badlav hai (Inclusivity ke liye).

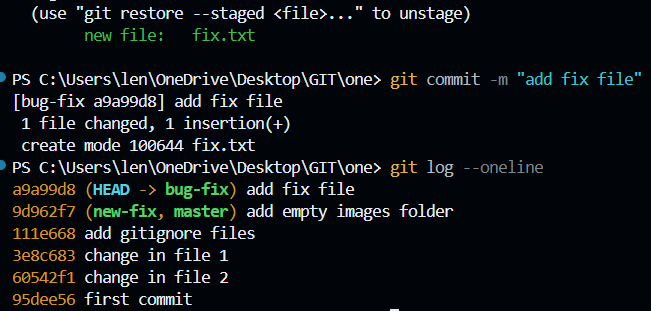




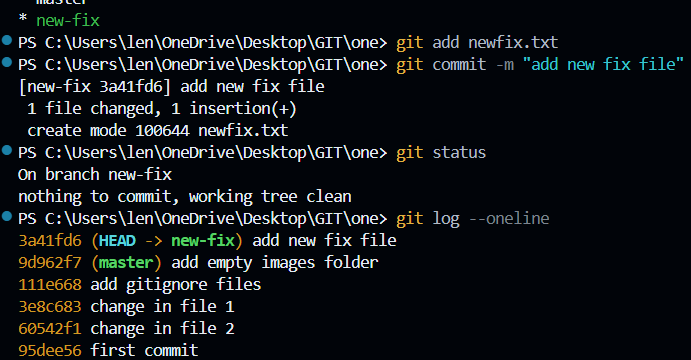


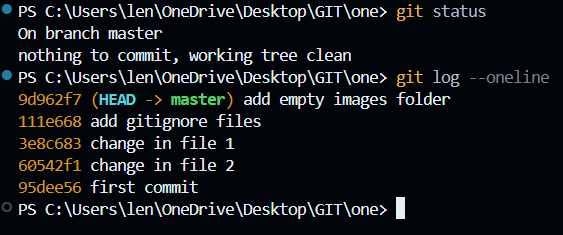


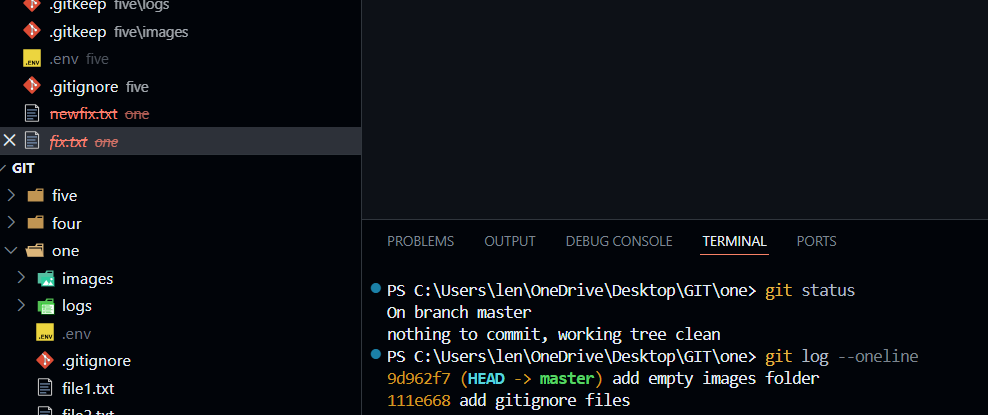
**AFTER CHANGES DONE IN BUG-FIX BRANCH**



Agar hamne koi nayi branch me switch kiya to purana wale branch ka jo naya file create kiya tha hamne wo hatt jayega kyuki wo sirf usi branch me exist karta hai..jabtak ki na ham usko merge karde master ya main branch se tabtak..!!







**MERGING**

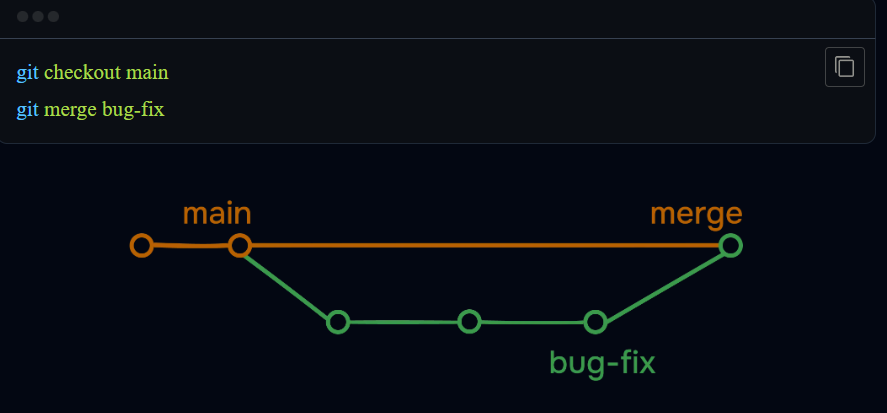
**Merging: Do raston ka milna**

Jab branch ka kaam khatam ho jaye, toh use main mein wapas laana hota hai. Iske do tarike hain:

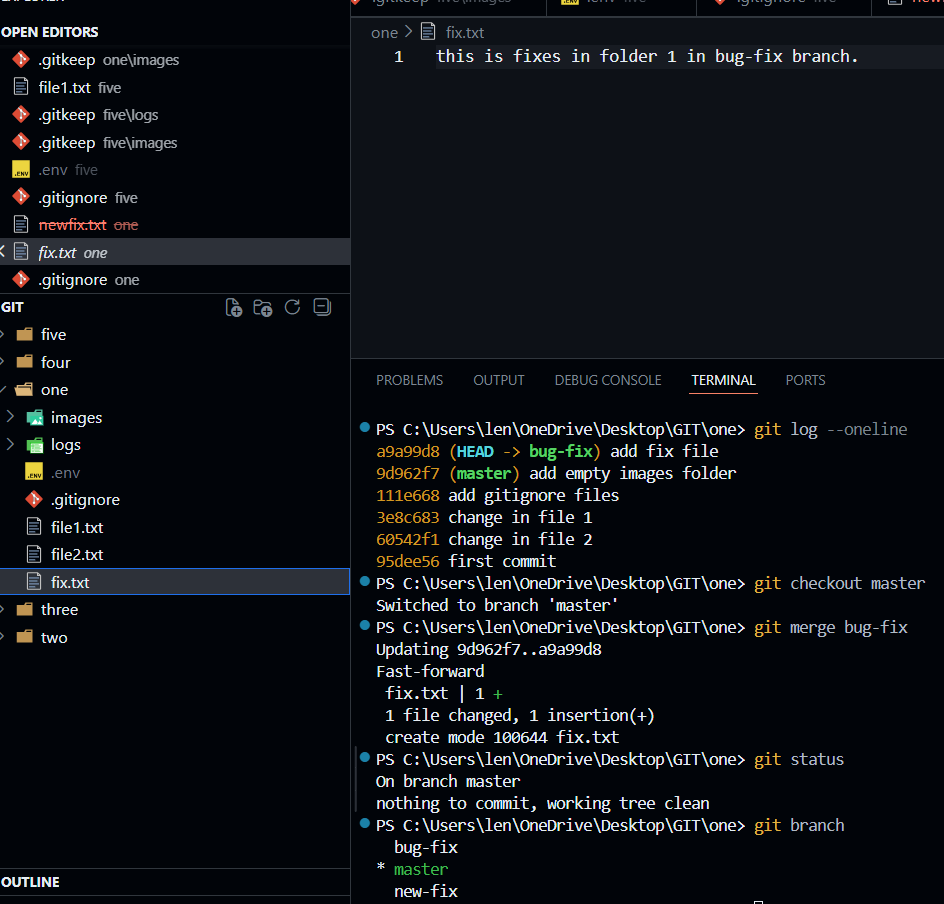
**A. Fast-Forward Merge (Easy Mode)**

Ye tab hota hai jab aapne main se branch banayi, usme kaam kiya, aur us beech main branch mein koi naya change **nahi** aaya.

* Git bas main ke pointer ko aage badha kar branch ke level par le aata hai. Koi tension nahi!

****

**This is simple fast forward merge..yaha pe master branch pe kaam nahi hua koi bhi jab bug-fix me kaam chal raha tha aur later on koi conflict ya commit ka issue nahi aaya merge karte samay master branch ke saath!!**

****

**B. 3-Way Merge (Diverged Paths)**

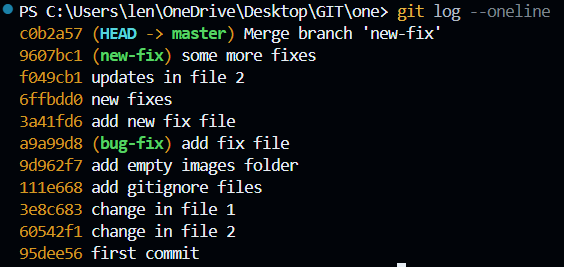
Ye tab hota hai jab aapne branch par kaam kiya, lekin us beech kisi aur ne main branch par bhi koi commit kar diya.

* Ab dono raste alag (diverge) ho gaye hain.
* Git in dono ko jodne ke liye ek **"Merge Commit"** banata hai. Isme 3 cheezein dekhi jati hain: (1) Purana common ancestor, (2) Branch ka naya kaam, aur (3) Main ka naya kaam.





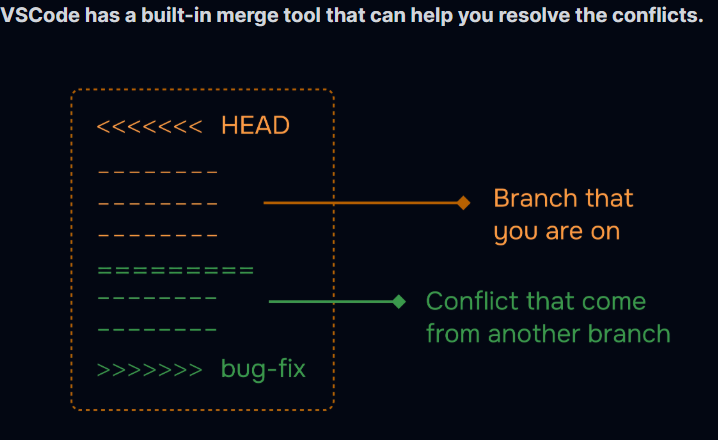
**Yaha pe jab new-fix branch pe kaam chal raha tha usi time master branch pe bhi kaam chal raha tha but alag alag files pe..(ye conflict case hatake bol raha)..so jab merge karenge unko to ek commit message VIM code editor me khulega..well VS code me hi rahega wo because of config –global core.editor “code --wait”..so uss time hame merge commit karna padega!!**

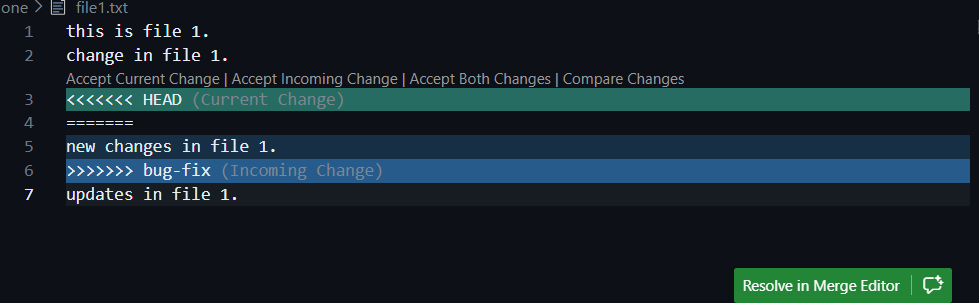


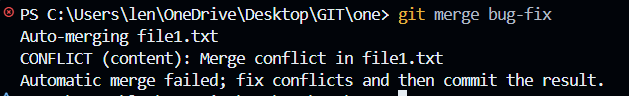
**Merge Conflicts:**

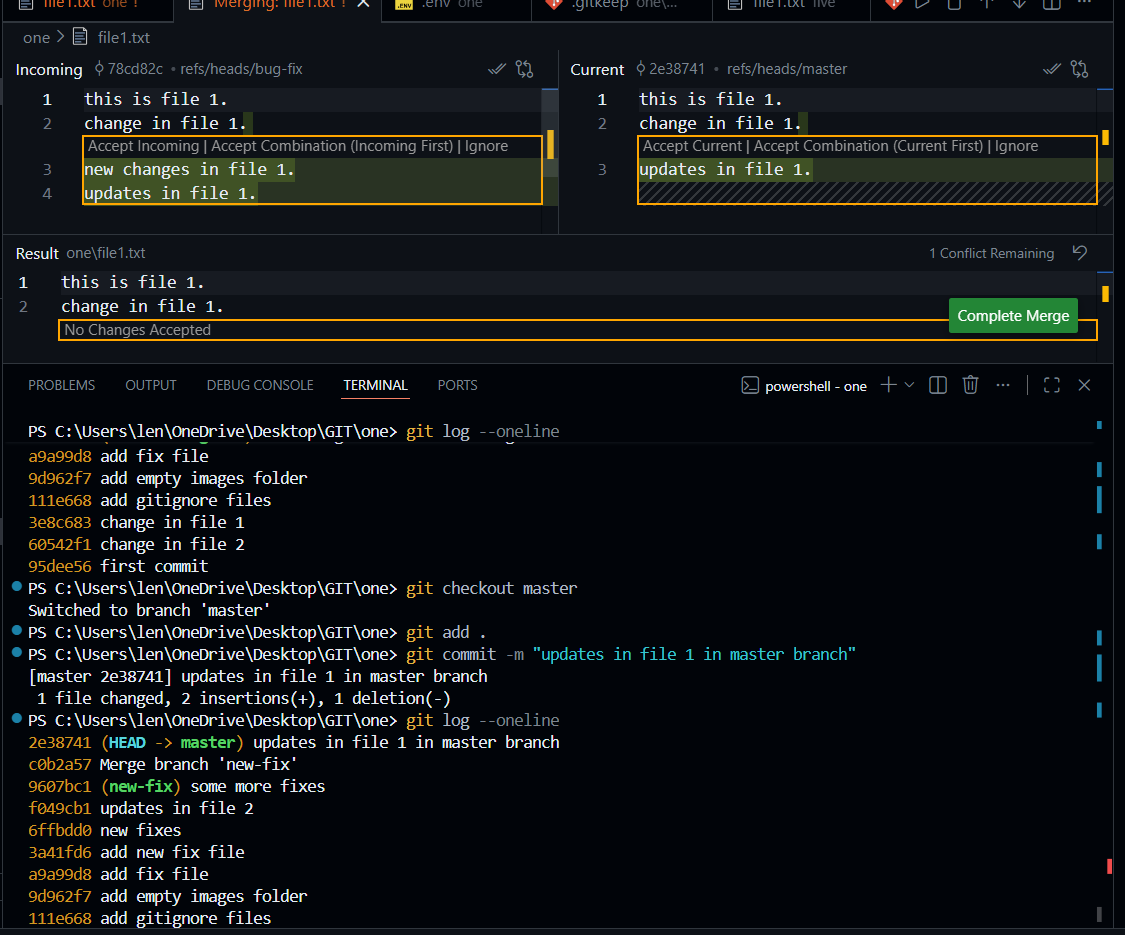
Jab do log ek hi file ki **ek hi line** ko change kar dete hain, toh Git confuse ho jata hai: *"Kiska rakhun? Tera ya tere dost ka?"* Ise hi **Conflict** kehte hain.

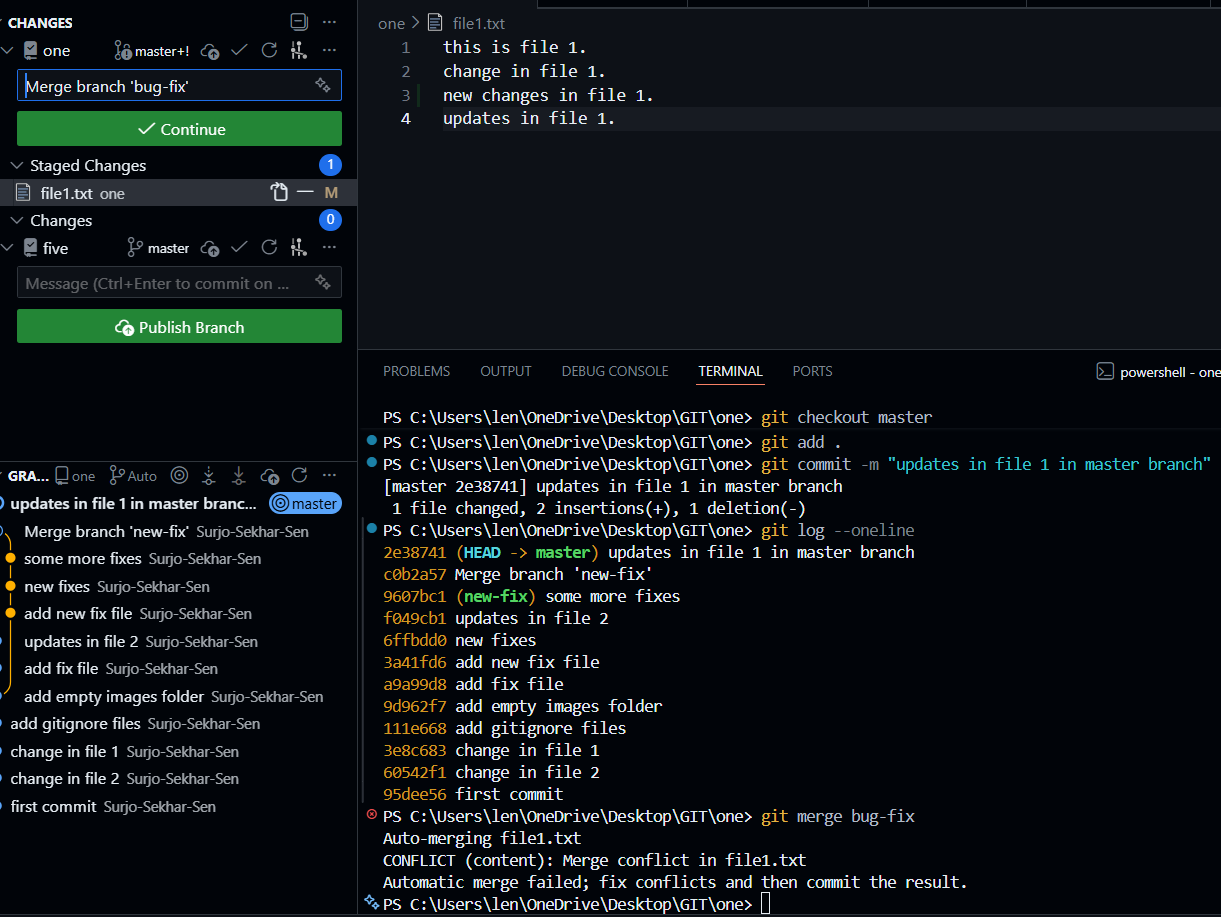
* **Kaise solve karein?** VS Code mein aapko options dikhte hain:
  + *Accept Current Change:* Mera wala rakho.
  + *Accept Incoming Change:* Dost wala rakho.
  + *Accept Both:* Dono ko upar-niche rakh do.
* Conflict solve karne ke baad aapko fir se git add aur git commit karna padta hai ye batane ke liye ki "Jhagda sulajh gaya hai!"



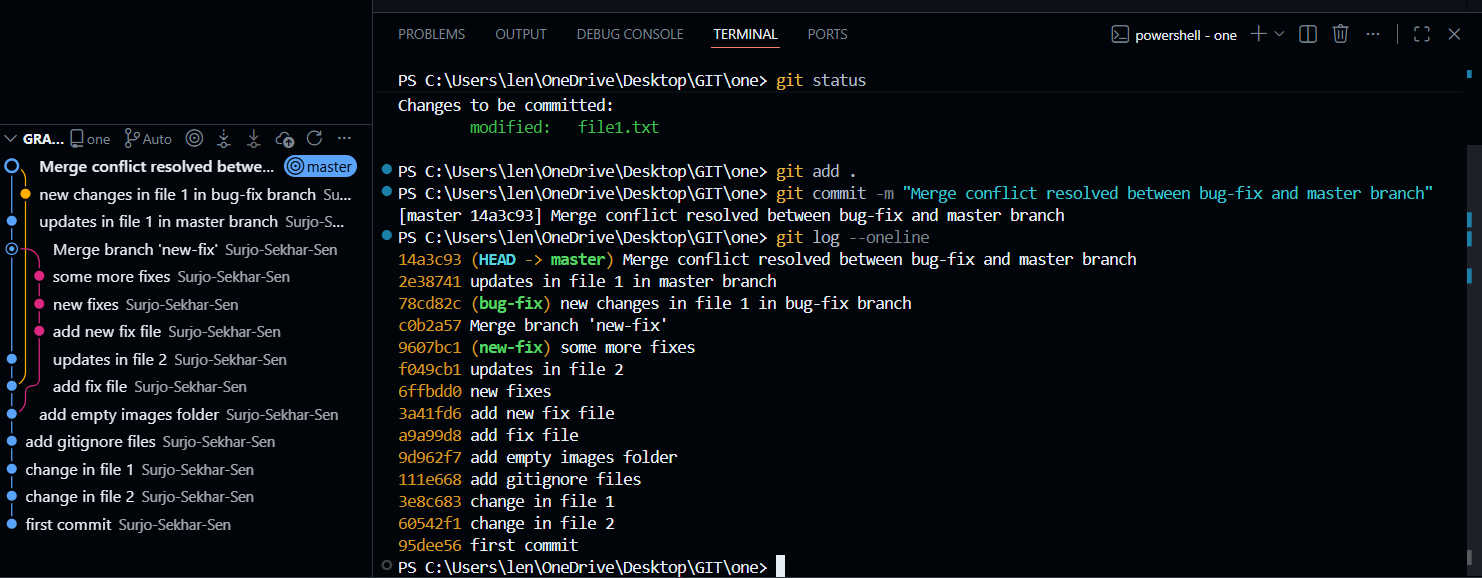








**Once this is done yani ki select karliya ki kaun sa choose karna hai tab hame commit karna hoga add karke.!!**



**IMPORTANT**

**(Q). Achha agar maine same folder ko liya aur usko maine git bash se ek khola aur changes kiye usme..and ek maine vscode ke terminal pe khola same folder aur kuch different changes kiye to kya wo same honge ya phir alag honge ya maane jayenge??**

**Wo changes "Same" hi maane jayenge**, kyunki Git aapke terminal ko nahi, balki **folder** ko track karta hai.

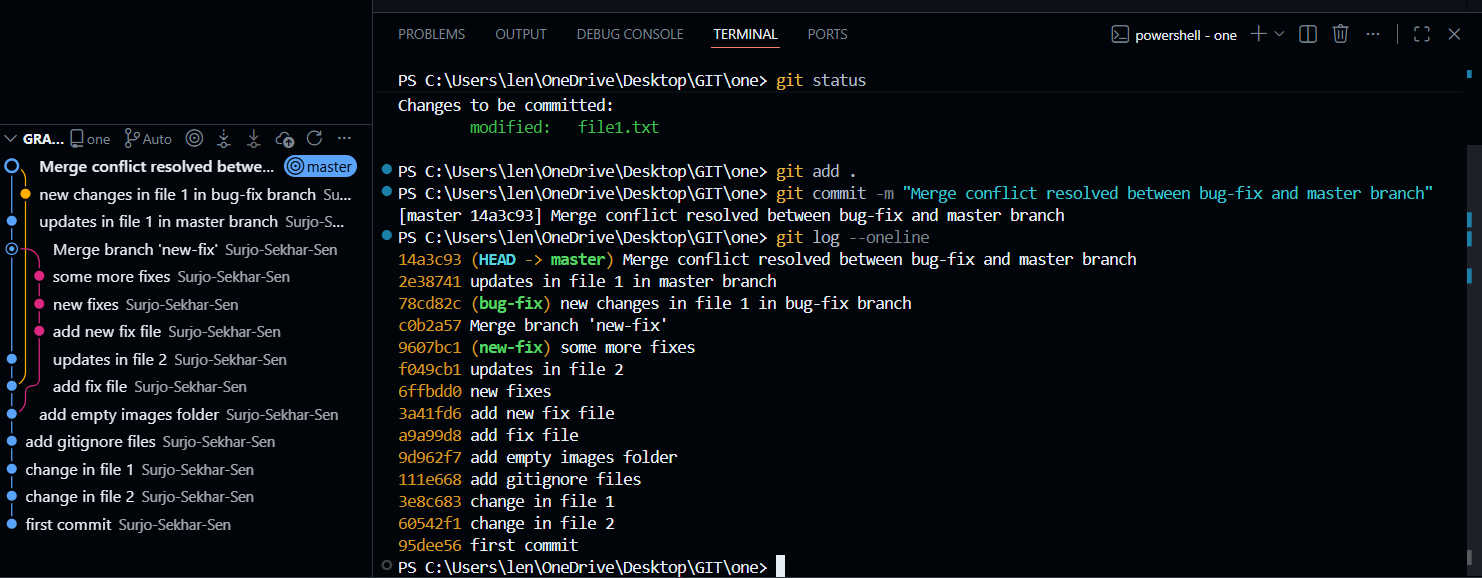
Isko thoda detail mein aur technical tarike se samajhte hain:

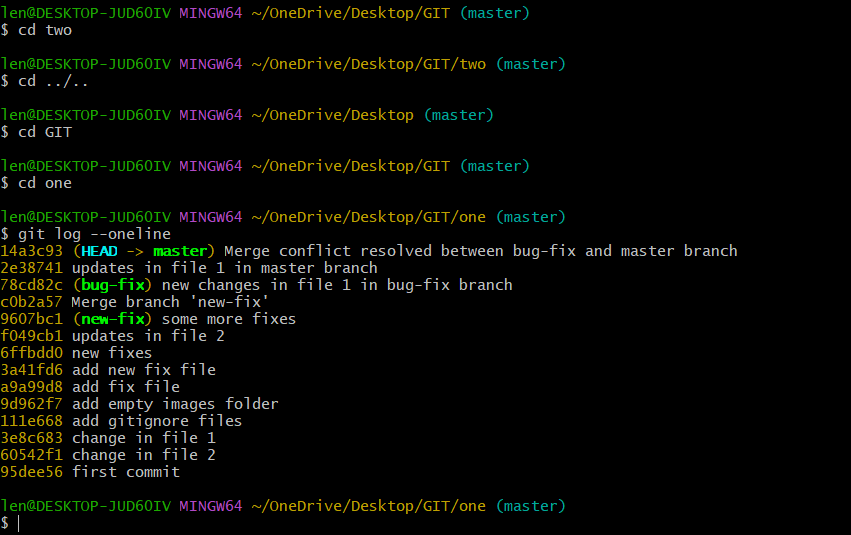
**1. Folder Ek Hai, Darwaaze Do Hain**

Socho aapka project ek "Kamra" (Room) hai.

* **Git Bash** ek khidki hai jahan se aap kamre ke andar dekh rahe ho.
* **VS Code Terminal** dusri khidki hai.

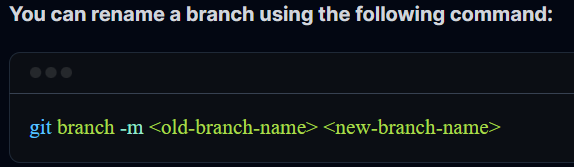
Agar aap Git Bash se kamre mein ek kursi (file) rakhte ho, toh jab aap VS Code wali khidki se dekhoge, toh wahan bhi kursi dikhegi. Kyunki asli badlav **Folder** ke andar ho raha hai, terminal mein nahi.

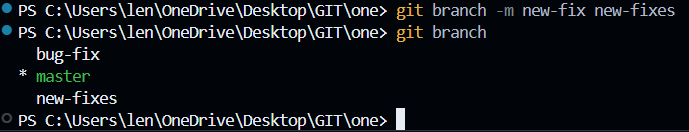




**GIT BRANCH RENAME**

**Rename (-m):** Agar spelling galat ho gayi branch ki.





**GIT BRANCH DELETE**

**Delete (-d):** Kaam hone aur merge karne ke baad kachra saaf karna zaroori hai. Faltu branches delete kar deni chahiye.



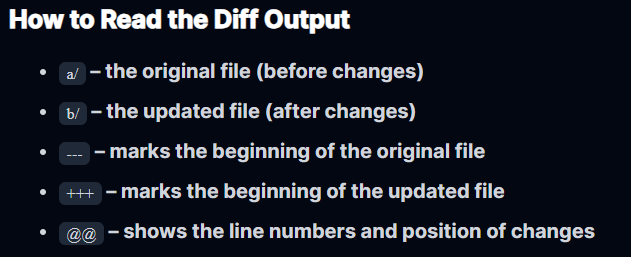


**GIT DIFF**

**1. Git Diff: "Farak Kya Hai?"**

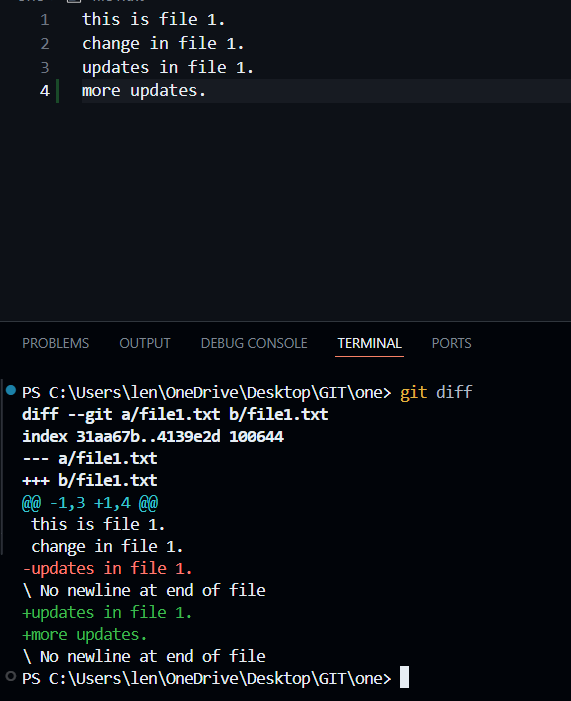
git diff ka asli kaam hai do cheezon ke beech ka antar (difference) dikhana. Jab aap ise terminal par chalate ho, toh ye kuch ajeeb dikhta hai (+++, ---, @@), par ise padhna simple hai:

* **Red Lines (-):** Ye wo lines hain jo delete hui hain ya purani hain.
* **Green Lines (+):** Ye wo lines hain jo aapne nayi add ki hain.
* **Context:** Git upar-niche ki 2-3 lines bhi dikhata hai taaki aapko pata chale change kahan hua hai.

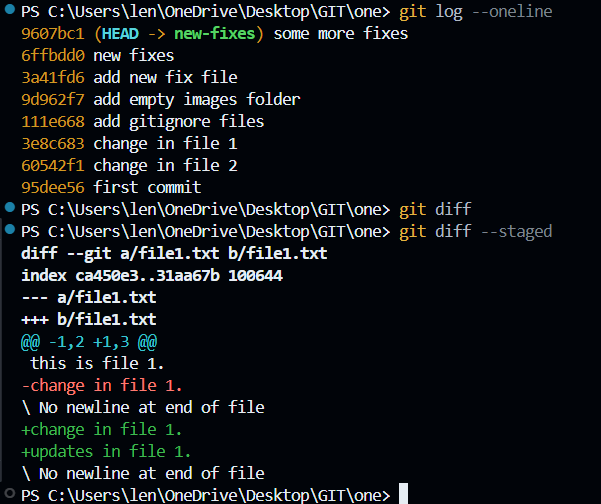


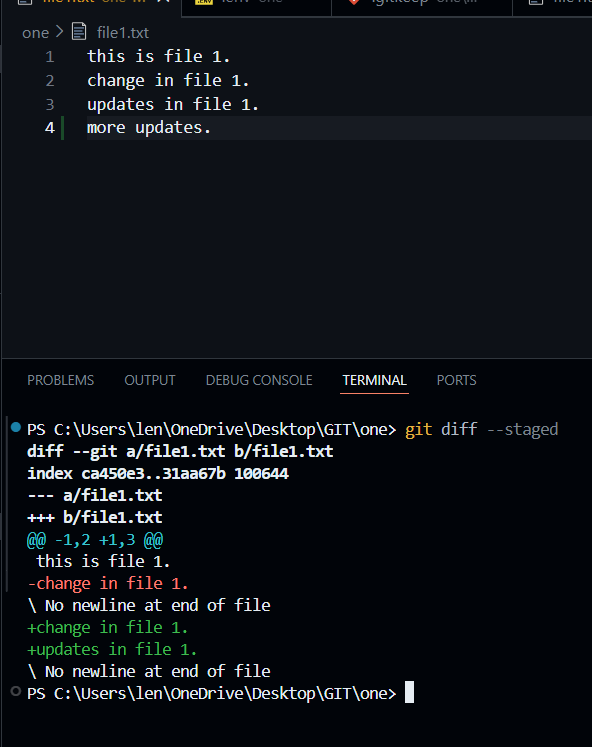
**Diff ke alag-alag roop:**

1. **git diff: Sirf un changes ko dikhayega jo aapne file mein kiye hain par abhi git add nahi kiye (Unstaged).**

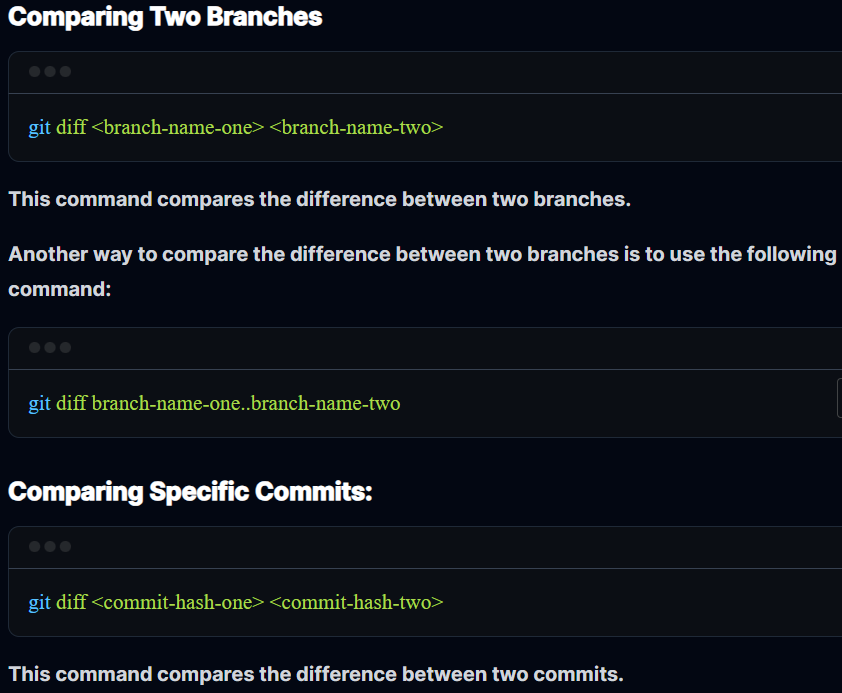


1. **git diff --staged: Ye un changes ko dikhayega jo aap git add kar chuke ho par abhi commit nahi kiya. (Commit karne se pehle double-check karne ka best tarika).**

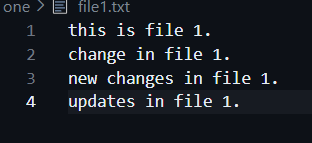
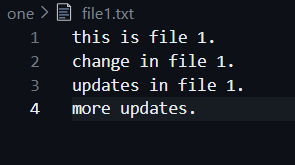


****

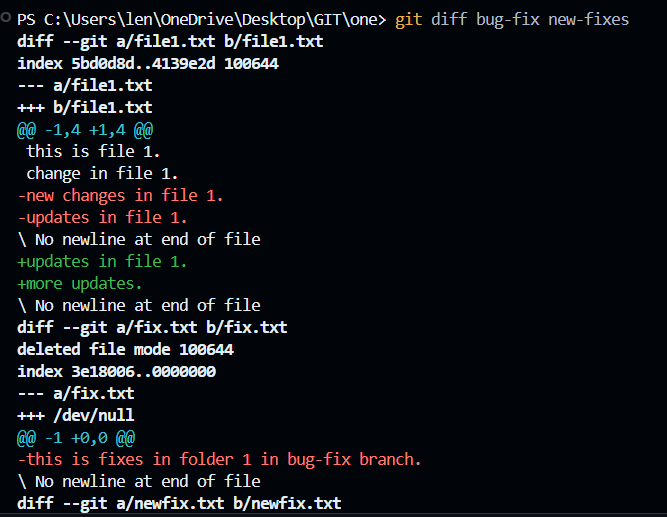
1. **git diff main..feature: Do alag raste (branches) ke beech ka farak dekhne ke liye.**



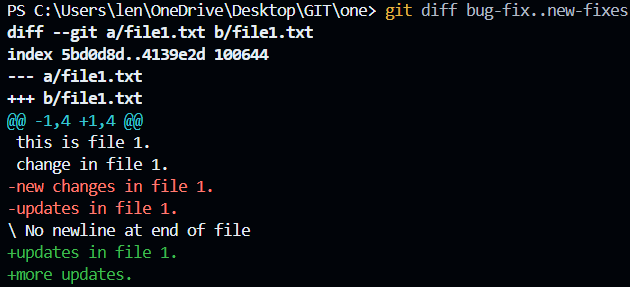
**BETWEEN BRANCHES**

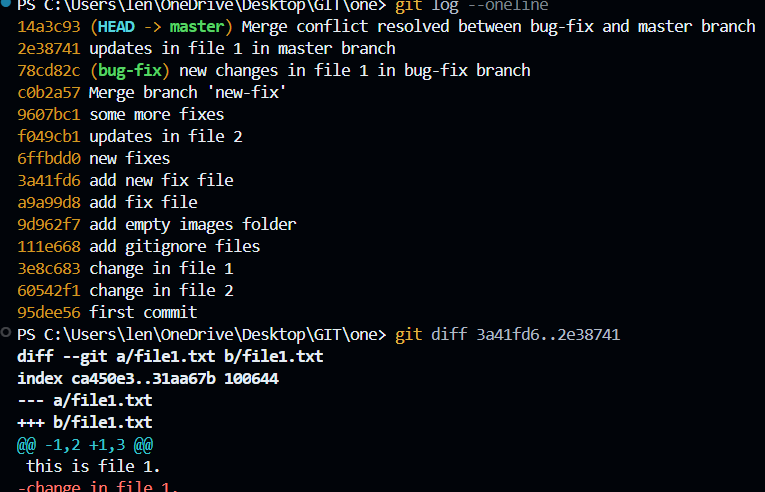
FILE 1 IN BUG-FIX BRANCH FILE 1 IN NEW-FIXES BRANCH

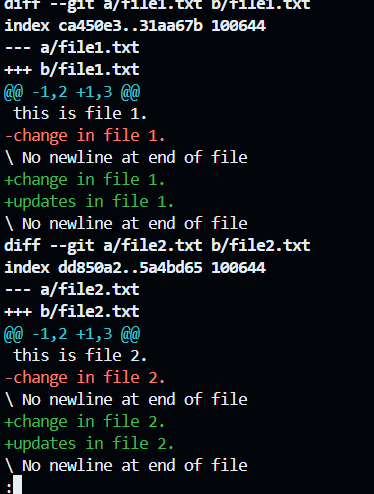


**OR A DIFFERENT WAY OF WRITING DIFF**



**BETWEEN HASH**





**GIT STASH**

**Git Stash: "Pausable Development"**

Stash ka matlab hota hai **"Chhupana"** ya **"Safe rakhna"**.

**Problem:** Maan lo aap ek feature par kaam kar rahe ho aur code abhi "adha-adhura" (broken) hai. Tabhi achanak boss ka call aata hai ki "Main branch par ek urgent bug fix karo". Aap adha code commit nahi kar sakte (kyunki wo broken hai) aur switch bhi nahi kar sakte (kyunki Git error dega).

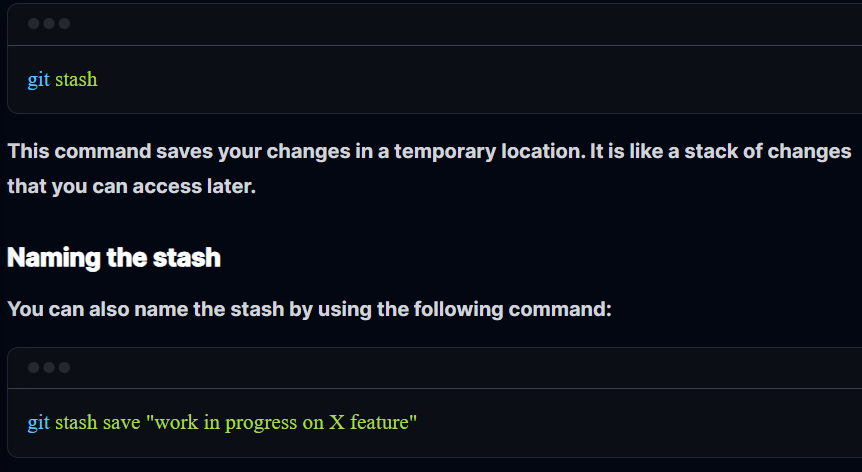
**Solution: git stash**

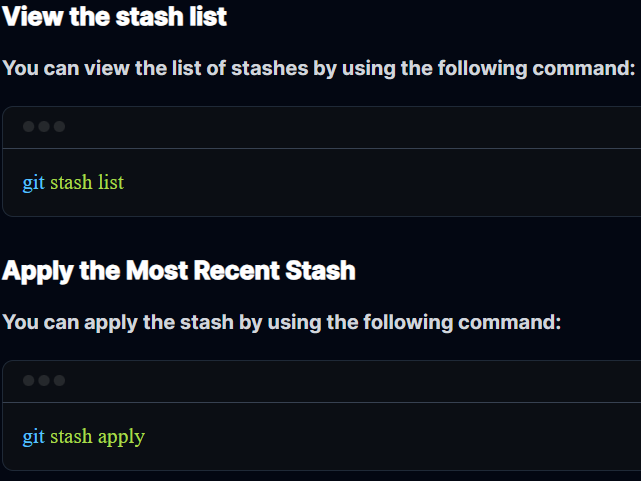
* Ye aapke current changes ko ek **Temporary Stack** (ek almari ki tarah) mein daal deta hai.
* Aapka working directory ekdum saaf ho jata hai.
* Jab aap bug fix karke wapas aate ho, toh **git stash pop** karo aur aapka adha-adhura kaam wapas aa jayega!

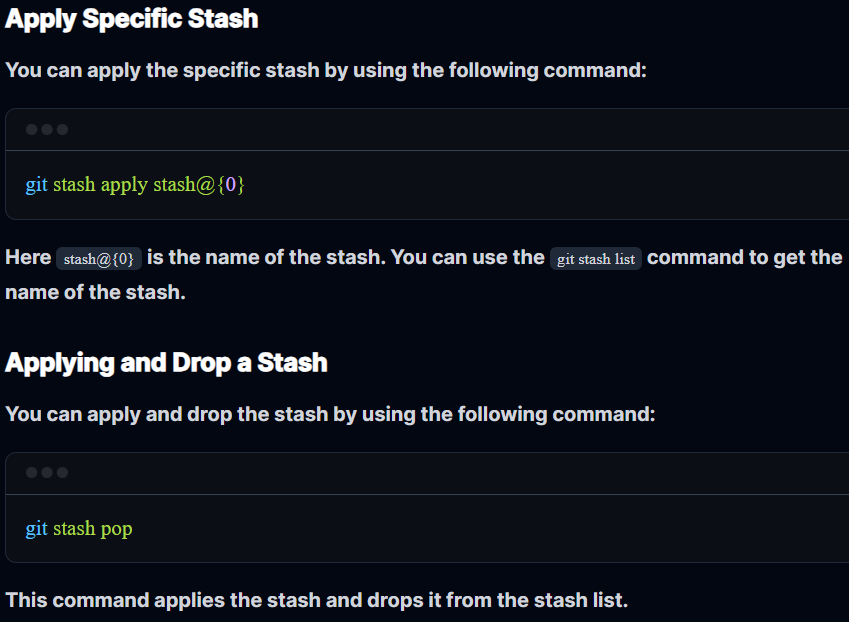
**Tip:** git stash pop changes ko wapas laakar stash list se delete kar deta hai. git stash apply changes laata hai par list mein copy rakhta hai.

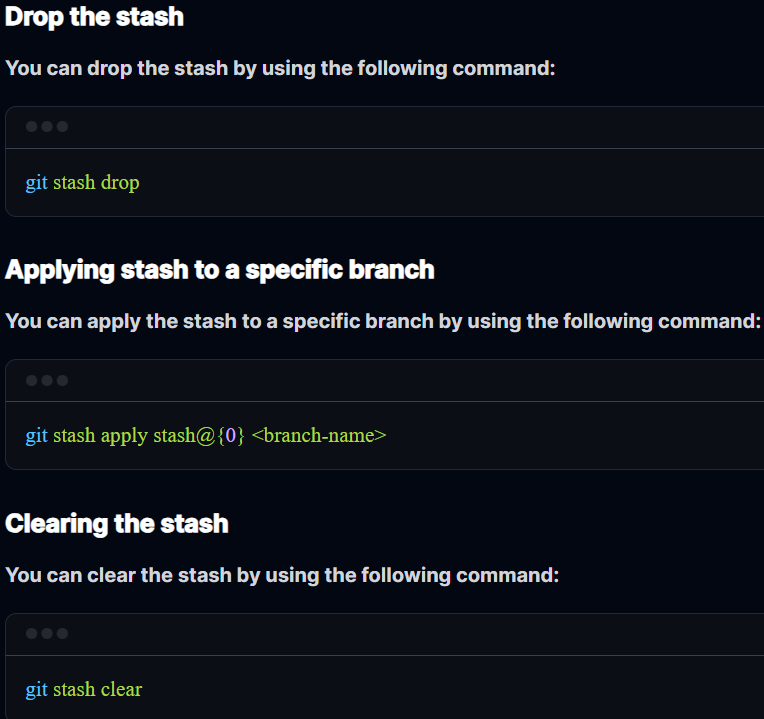
**JAB AISA KOI SITUATION AA JAYE JAHA HAM EK BRANCH ME HAI AND ABHI TAKK COMMIT YA ADD NAHI KIYA KYUKI HAM KUCH DUSRE BRANCH ME CHECK KARNA CHAHTE HAI TO HAM USSKO STASH KARKE DUSRE BRANCH PE JO KAAM HAI WO KARKE AAKE STASH KO APPLY KARKE WAPAS SE KAAM KAR SAKTE HAI CHAHE ADD YA COMMIT.!!**

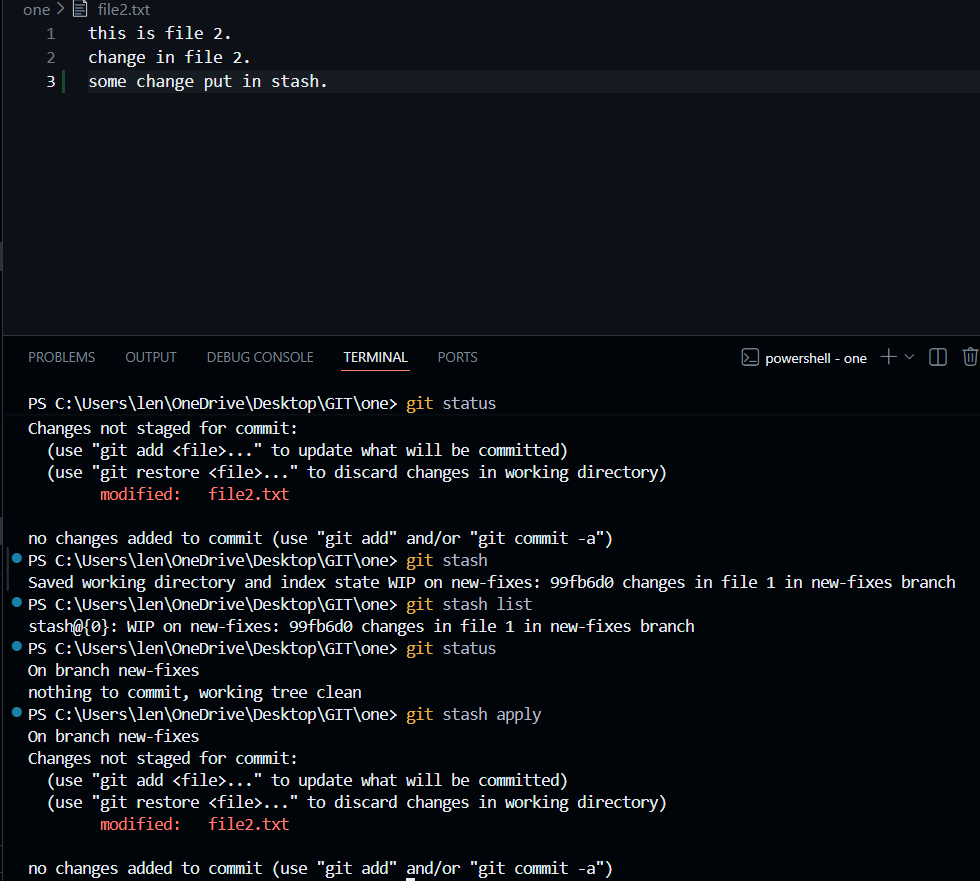


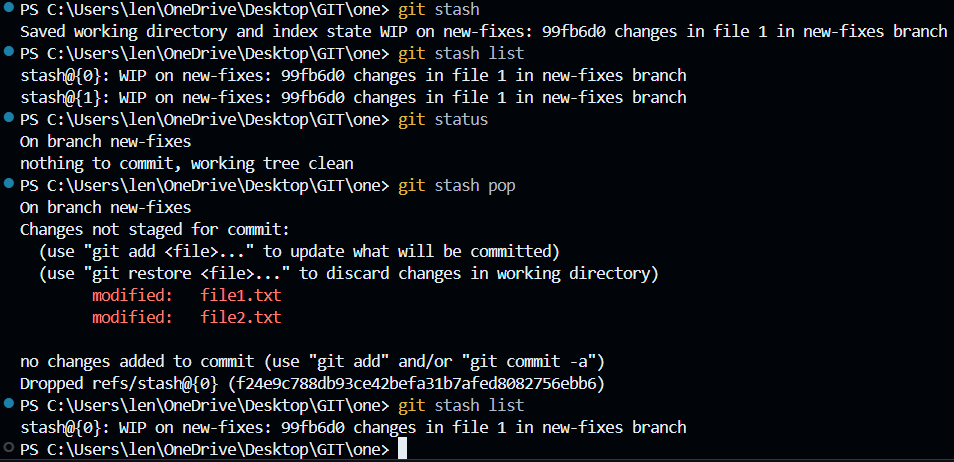








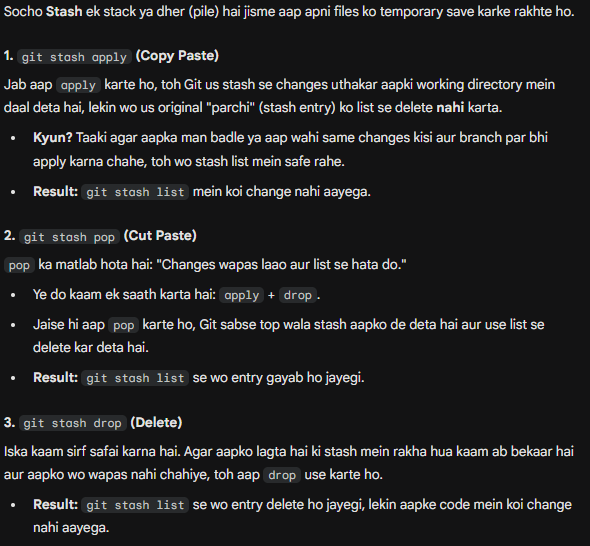




**IMPORTANT:**

**JAB BHI STASH APPLY KARTE HO TO WO TUM KISI BHI BRANCH ME KAR SAKTE HO AGAR SAME BRANCH ME KAR RAHE JAHA PAR STASH KIYA THA TO KOI CONFLICT NAHI AAYEGA LEKIN AGAR KISI DUSRE BRANCH ME JAAKE APPLY KAR RAHE HO TAB MAYBE CONFLICT AA SAKTA HAI.!!**

**JAB BHI STASH POP KAR RAHE HO AND STASH LIST CHECK KAR RAHE HO..STASH LIST SE ITEM SIRF TABHI BAHAR JAYEGA AGAR STASH KO POP KIYA GAYA HO JIS BRANCH ME TUMNE USKO STASH KIYA THA WAHA SE POP YA DROP KARNE PARR HI WO LIST SE HATEGA NAHI TO WAHI RAHEGA.!!**



**GIT TAG**

**Git Tags: "Milestone Markers"**

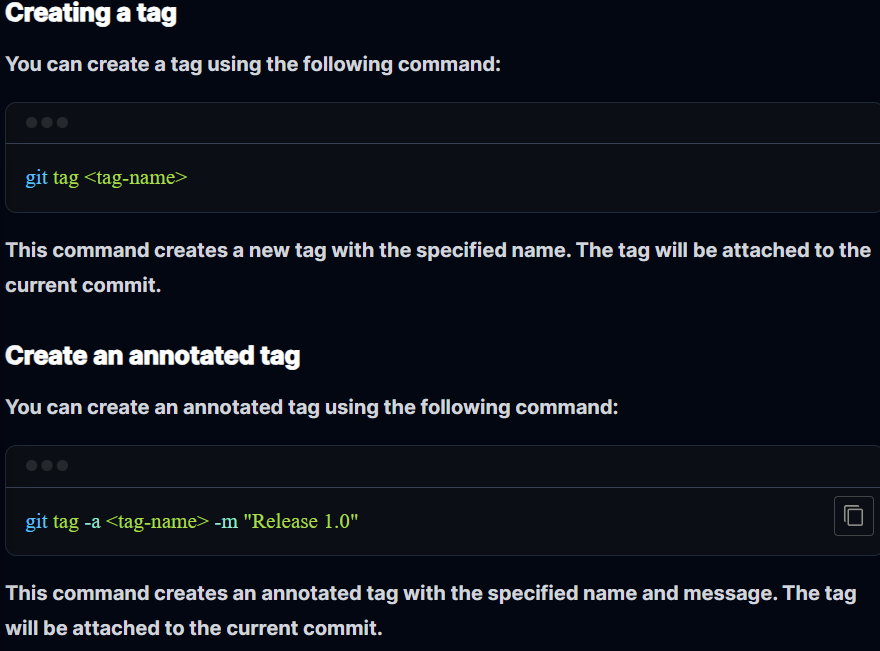
Agar commit ek "Save Point" hai, toh **Tag** ek "Special Milestone" hai.

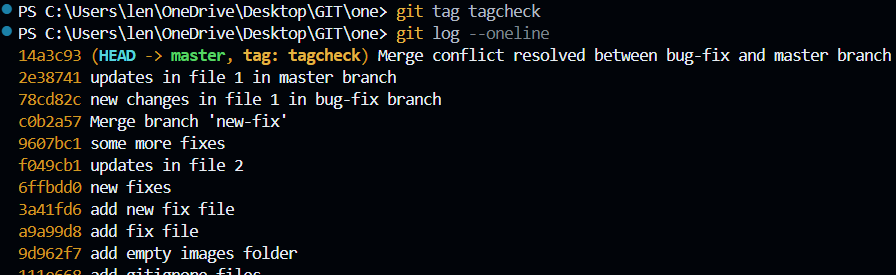
**Kyun use karein?** Ek project mein hazaron commits ho sakte hain. Lekin aapko kaise yaad rahega ki kaunse commit par aapne **v1.0 (Version 1)** launch kiya tha?

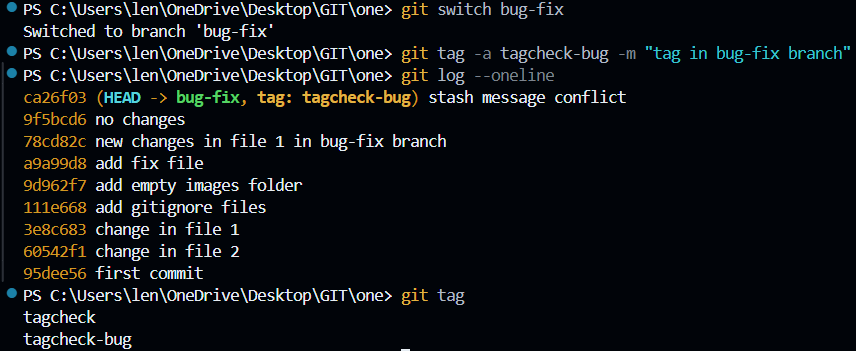
* Aap us commit par ek **Tag** laga dete ho (jaise sticky note).
* Ye hamesha ke liye fixed rehta hai. Branch aage badhti rahegi, par Tag wahi rahega.

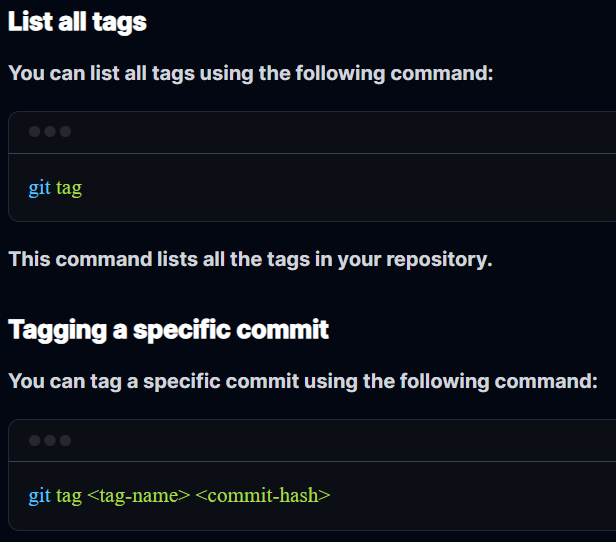
**Types of Tags:**

1. **Lightweight:** Sirf ek naam (Jaise: v1.0).
2. **Annotated (-a):** Isme tag lagane wale ka naam, date aur ek message bhi hota hai. Production releases ke liye hamesha **Annotated tags** hi use karne chahiye.

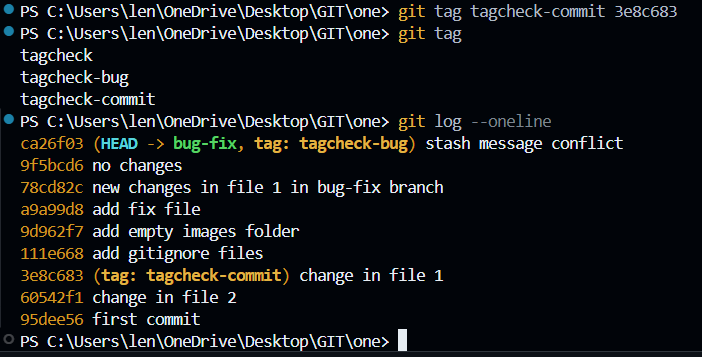




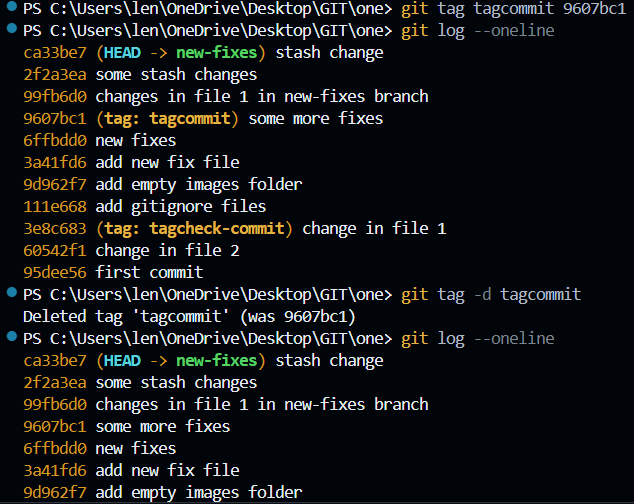


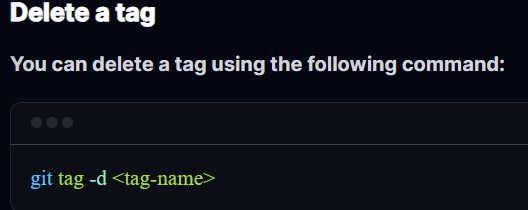


**GIT TAG LIST AND COMMIT ID**

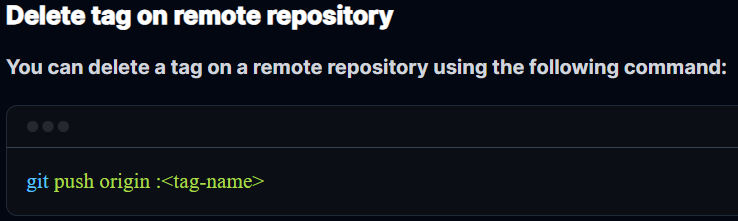


**GIT TAG DELETE**









**IMPORTANT**

**REMOTE REPOSITORY**

**Remote Repository** (Remote Repo) ka matlab hai aapke project ki woh copy jo internet par kisi server par host ki gayi hai, jaise **GitHub, GitLab, ya Bitbucket** par.

Abhi tak aap jo bhi git init, add, ya commit kar rahe the, woh sab aapke laptop (**Local Repo**) ke andar ho raha tha. Agar aapka laptop kharab ho jaye, toh aapka sara kaam gaya! Isliye hum Remote Repo ka use karte hain.

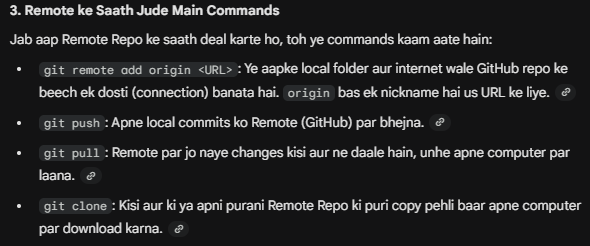
Isko detail mein samajhte hain:

**1. Local vs Remote: Asli Farq**

* **Local Repository:** Aapke computer ki hard drive par hoti hai. Aap offline kaam kar sakte ho, commit kar sakte ho.
* **Remote Repository:** Ek cloud server par hoti hai. Ye pure team ke liye ek **"Central Hub"** ka kaam karti hai.

**2. Remote Repo Kyun Zaroori Hai?**

1. **Backup & Safety:** Agar aapka computer crash ho gaya, toh aap kisi bhi naye machine par apna pura code wapas paa sakte ho.
2. **Collaboration (Team Work):** Maan lo aap Bangalore mein ho aur aapka dost Delhi mein. Aap dono ek hi project par kaam kaise karoge? Aap apna code Remote par **Push** karoge, aur aapka dost wahan se use **Pull** kar lega.
3. **Deployment:** Aaj kal zyadatar websites Remote Repo se hi utha kar live (internet par) ki jaati hain.



**GIT REBASE**

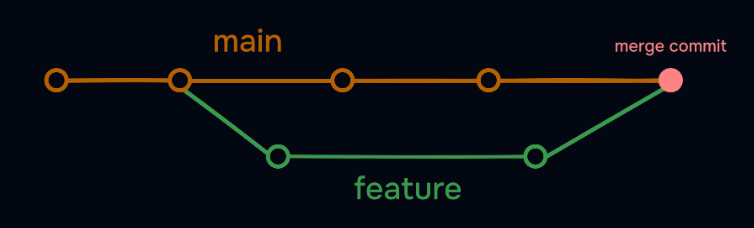
**1. Rebase: "History ko Linear banana"**

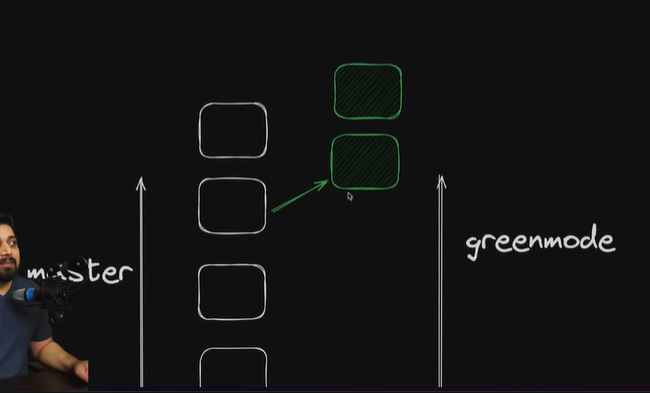
Rebase ka matlab hota hai **"Base badal dena"**.

**Merge aur Rebase mein farak kya hai?**

* **Merge:** Jab aap do branches ko jodte ho, toh Git ek naya "Merge Commit" banata hai. Isse history thodi messy dikhadi deti hai (bahut saari lines aapas mein judti hui).

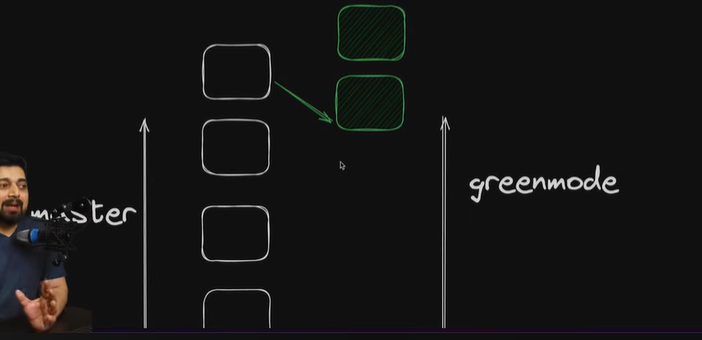
**USUAL FLOW AND MERGE COMMIT**





* **Rebase:** Ye kya karta hai ki aapke feature-branch ke saare commits ko uthata hai, aur unhe main branch ke sabse naye commit ke **upar** "paste" kar deta hai. Isse history aisi dikhti hai jaise saara kaam ek hi seedhi line (linear) mein hua ho.

**IF REBASE IS DONE**



**Caution:** Kabhi bhi us branch par rebase mat karo jo aapne GitHub par share kar di hai (Public branches). Isse doosre developers ki history bigad jayegi.

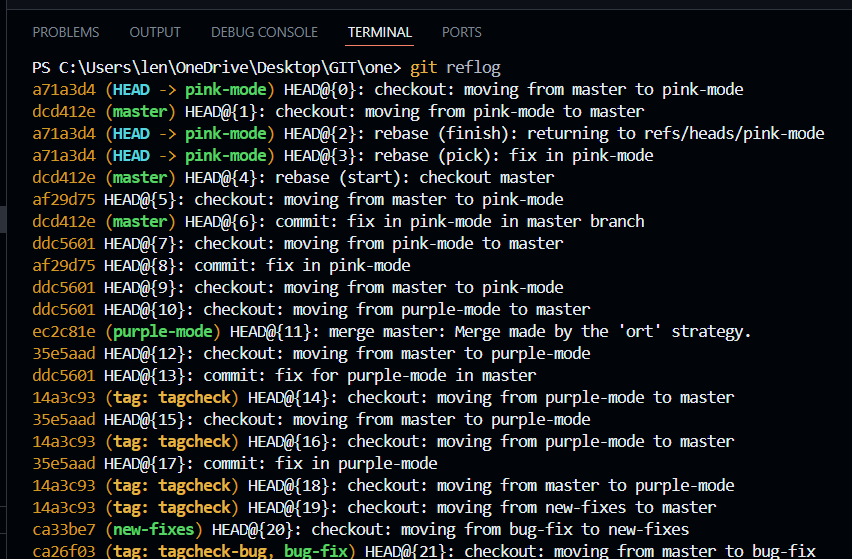
**ALSO KABHI BHI MAIN BRANCH PE REBASE MAT KARO!!**

**2. Reflog: "Git ki Secret Diary"**

Aksar log darte hain ki *"Maine galti se branch delete kar di, mera code gaya!"* Lekin Git bahut smart hai.

**Git Reflog** ek aisi jagah hai jahan Git har ek move ko record karta hai—chahe aapne commit kiya ho, branch switch ki ho, ya rebase kiya ho.

* git log sirf "safal" commits dikhata hai.
* git reflog woh sab dikhata hai jo aapne terminal par "chul" machayi hai.

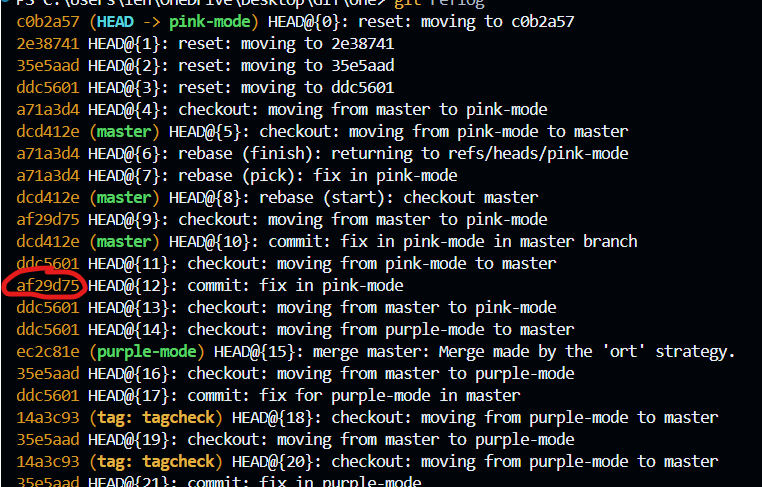


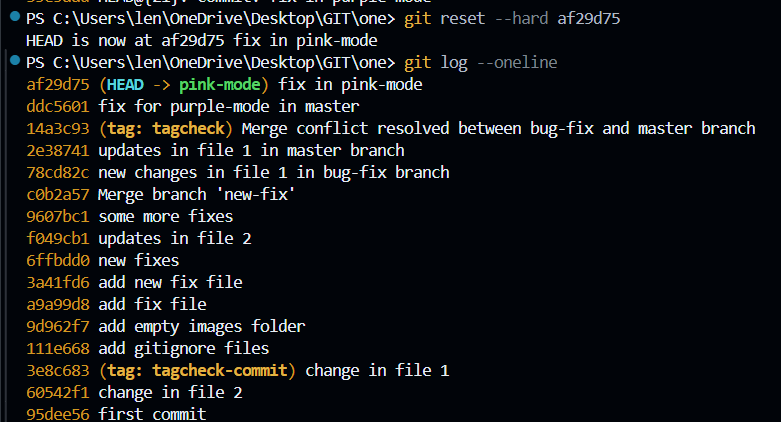
**3. Recovering Lost Work (The Magic Trick)**

Maan lo aapne galti se git reset --hard kar diya aur aapka pichla commit gayab ho gaya. Ab kya karein?

1. **git reflog** chalao.
2. Wahan dekho ki galti karne se theek pehle HEAD kahan tha (Jaise HEAD@{1}).
3. Command chalao: git reset --hard HEAD@{1}.

**Boom!** Aapka "deleted" code wapas aa gaya. Ye ek developer ki life mein kisi chamatkar se kam nahi hai.



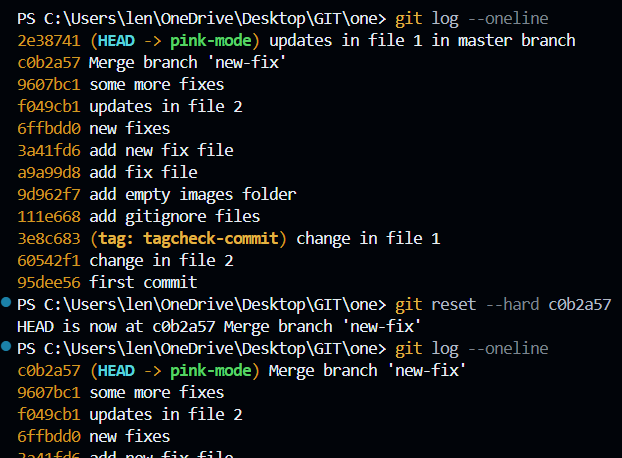


**ALL THE COMMITS WHICH GOT DELETED IN LOG CAME BACK!!**

**4. Rebase Flow with Conflicts**

Jab aap rebase karte ho aur conflict aata hai, toh Git ruk jata hai.

1. Pehle VS Code mein conflict fix karo.
2. Files ko git add karo.
3. Phir git commit **nahi** karna hai! Aapko bolna hai: **git rebase --continue**. Git agla commit uthayega aur use naye base par lagane ki koshish karega.



**Key Terms to Remember:**

* **Linear History:** Ek seedhi line mein commits ka hona (Rebase se hota hai).
* **Rewriting History:** Rebase commits ki ID badal deta hai, isliye ise "rewriting" kehte hain.
* **Hard Reset:** git reset --hard sab kuch mita deta hai, isliye ise sambhal kar use karein (par Reflog se bachaya ja sakta hai).

**Ek Pro Tip:** --force option ko tab tak use na karein jab tak aap 100% sure na hon. Iska naam hi hai "dhakka dena", aur ye aksar project ki history kharab kar deta hai.

**IMPORTANT**

**REFLOG VS LOG**

Jab aap git reset --hard <commit-id> karte ho, toh aapko lagta hai ki baaki ke commits hamesha ke liye delete ho gaye, lekin Git unhe turant delete nahi karta. Woh unhe "Reflog" ki memory mein 30-90 din tak rakhta hai.

Isko breakdown karke samajhte hain ki Reflog ID kaise kaam karti hai:

**1. Reflog ID kya hai? (HEAD@{n})**

Reflog ID koi static ID nahi hai, ye ek **"Time-Stamp"** ya **"Movement Record"** hai.

* HEAD@{0} ka matlab hai: Abhi main jahan hoon.
* HEAD@{1} ka matlab hai: Isse theek pehle main jahan tha.
* HEAD@{5} ka matlab hai: 5 moves pehle main jahan tha.

**2. Kya farak padta hai Reflog ID dene se?**

Jab aap git reset --hard HEAD@{1} dete ho, toh aap Git ko commit ID nahi, balki **"Ek kadam piche"** jaane ko bol rahe ho.

**Example Scenario:**

1. Aap **Commit A** par the.
2. Aapne **Commit B** kiya.
3. Galti se aapne git reset --hard Commit-A-ID chala diya. (Ab git log mein Commit B gayab ho gaya).
4. Ab aap git reflog dekhoge, toh wahan dikhayega ki:
   * HEAD@{0}: reset: moving to Commit A
   * HEAD@{1}: commit: Commit B

Ab agar aap git reset --hard HEAD@{1} karoge, toh aap **Commit B** par wapas pahunch jaoge!

**3. Commit ID vs Reflog ID**

Aap dono mein se kuch bhi de sakte ho:

* Agar aapko us "khoye huye" commit ki **Hash ID** (jaise abc1234) mil gayi hai reflog mein, toh aap woh use kar lo: git reset --hard abc1234.
* Agar aapko ID lambi lag rahi hai, toh short-cut mein HEAD@{n} use kar lo. Dono ka result **bilkul same** hoga.

**4. Ek important baat: git log vs git reflog**

* **git log**: Ye sirf "zinda" branches ke commits dikhata hai. Reset ke baad yahan se data gayab ho jata hai.
* **git reflog**: Ye Git ke har ek action ka record hai. Reset karne ke baad bhi purana data yahan dikhta hai.

**Moral of the story:** Reflog ID dene se "sab kuch" hota hai. Ye aapki galti sudhaarne ka aakhri raasta hai.

**PUBLIC AND PRIVATE SSH KEY FOR GIT AND GITHUB**

**Step 1: Nayi SSH Key Generate Karna**

Sabse pehle aapko apne computer par ek digital "lock" aur "key" banani hai.

1. **Git Bash** open karein.
2. Niche wali command paste karein (apna GitHub wala email use karein):

Bash

ssh-keygen -t ed25519 -C "your\_email@example.com"

* + **Note:** Agar aapka system purana hai, toh -t rsa -b 4096 use karein, warna ed25519 best aur secure hai.

1. **Enter press karein:** Jab wo puche "Enter file in which to save", toh bas **Enter** daba dein (wo default folder /c/Users/YOU/.ssh/ mein save ho jayega).
2. **Passphrase (Optional):** Ye ek extra password hota hai. Agar aap baar-baar password nahi dalna chahte, toh ise **khaali chhod dein (sirf Enter daba dein)** do baar.

**Step 2: SSH Agent ko Start karna aur Key Add karna**

Sirf key banana kaafi nahi hai, computer ko batana padta hai ki "Bhai, ye rahi meri key, ise use karna."

1. **PowerShell** ko "Run as Administrator" karke kholiye aur ye commands chalaiye (taaki background service start ho jaye):

PowerShell

Get-Service -Name ssh-agent | Set-Service -StartupType Manual

Start-Service ssh-agent

1. Ab apni key ko agent mein add karein (Normal terminal se bhi kar sakte hain):

Bash

ssh-add ~/.ssh/id\_ed25519

*(Agar aapne key ka naam kuch aur rakha tha, toh wahi naam yahan likhein).*

**Step 3: GitHub par Public Key Add karna (Sabse Zaruri)**

Ab aapko "Lock" (Public Key) GitHub ko dena hai.

1. Apni public key ko copy karein. Terminal mein ye command chalao:

Bash

cat ~/.ssh/id\_ed25519.pub

Jo text niche dikhega (shuruwat ssh-ed25519... se hogi), use pura **copy** kar lo.

1. **GitHub Website** par jao:
   * Settings -> SSH and GPG keys -> **New SSH Key**.
2. **Title** mein apne laptop ka naam rakh do (e.g., "My Laptop").
3. **Key** wale box mein wo pura text **paste** kar do jo copy kiya tha.
4. **Add SSH Key** par click karo. Done!

**Key Terms Jo Aapne Padhe:**

* **Passphrase:** Key ke upar ek extra lock. Agar aapne set kiya, toh har baar code push karte waqt ye maangega. Isliye naye log ise aksar blank chhod dete hain.
* **ssh-agent:** Ek background program jo aapki keys ko "yaad" rakhta hai taaki aapko baar-baar setup na karna pade.
* **Hardware Security Key:** Ye ek physical USB device hoti hai (jaise YubiKey). Agar aapke paas ye nahi hai, toh upar wala standard method hi best hai.

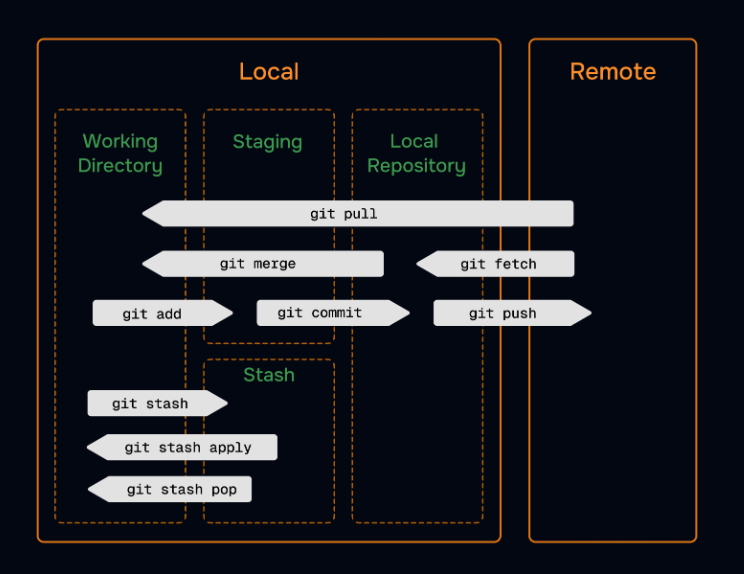
**Kaise Check Karein ki Sab Sahi Hai?**

Terminal mein ye command chalao:

Bash

ssh -T git@github.com

Agar aapko message mile: *"Hi [YourUsername]! You've successfully authenticated..."*, toh samajh lo aap **Git Ninja** ban gaye ho! 🥷



**1. SSH Key: Aapke Computer ka Digital ID Card**

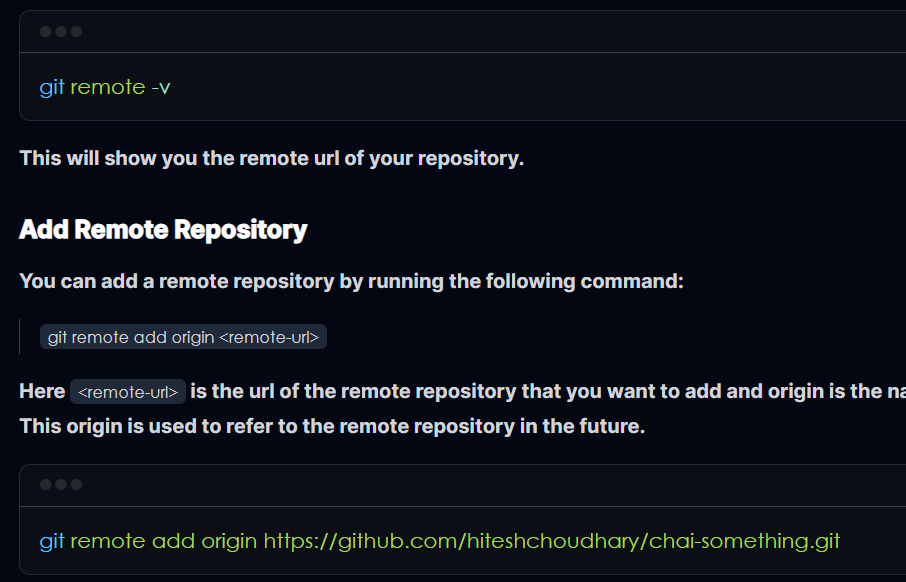
Pehle GitHub password maangta tha, lekin ab security ke liye **SSH Key** zaroori hai. Iska faida ye hai ki aapko baar-baar login nahi karna padta.

* **ssh-keygen**: Ye command aapke liye do keys banati hai.
  + **Private Key:** Ye aapke computer ke andar chhupi rehti hai (ise kabhi kisi ko mat dikhana).
  + **Public Key (.pub):** Ise aap GitHub par "Add SSH Key" wale section mein jaakar paste karte ho.
* Jab aap code push karte ho, toh GitHub aapki Public Key aur aapke computer ki Private Key ko match karta hai. Agar match ho gaya, toh entry mil jati hai.

**2. Remote Repo se Connection Jodna**

Jab aap GitHub par naya repo banate ho, toh wo khali hota hai. Local repo ko usse jodne ke liye hum git remote command use karte hain.

* **git remote add origin <URL>**:
  + **origin**: Ye bas ek nickname hai. Aap URL ki jagah origin bol sakte ho.
  + **URL**: Ye aapke GitHub repo ka address hota hai.

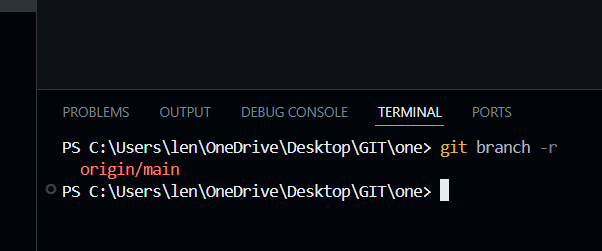


**3. Pushing Code: Local se Cloud tak**

git push origin main ka matlab hai: *"Mere main branch ke saare commits origin (GitHub) par bhej do."*

* **-u flag ka jaadu:** Jab aap pehli baar git push -u origin main karte ho, toh -u (upstream) local aur remote branch ke beech ek pakka rishta bana deta hai.
* Iske baad aapko sirf git push ya git pull likhna padega, Git apne aap samajh jayega ki kahan bhejna hai.

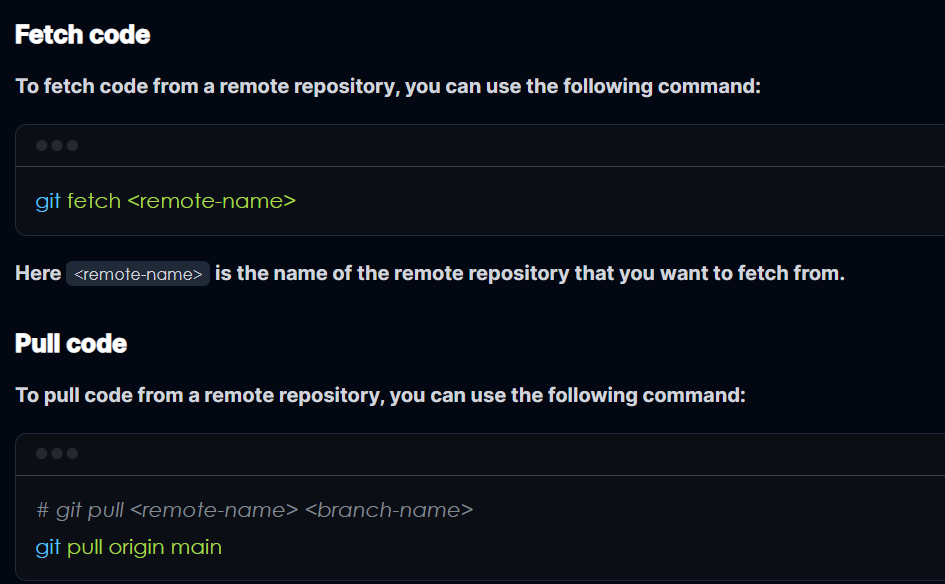
**TO CHECK BRANCHES IN REMOTE REPO**

****

**4. Fetch vs Pull: Sabse bada confusion**

Dono ka kaam remote se code lana hai, par ek chota sa farak hai:

* **git fetch**: Ye sirf naya code download karta hai par aapke kaam mein **"Merge"** nahi karta. Ye sirf aapko dikhata hai ki GitHub par kya naya hai. (Safety first!)
* **git pull**: Ye **Fetch + Merge** ka combo hai. Ye naya code download bhi karega aur turant aapke local code ke saath jod (merge) dega.



**5. Summary Flow (First Time Setup)**

Agar aap pehli baar project GitHub par daal rahe ho, toh ye sequence yaad rakho:

1. **git init** (Local repo banao)
2. **git add .** (Files stage karo)
3. **git commit -m "initial commit"** (Save point banao)
4. **git remote add origin <URL>** (GitHub se jodo)
5. **git push -u origin main** (Code bhej do!)

**Ek Zaroori Tip: Upstream vs Origin**

* **Origin:** Ye aapka apna repo hai GitHub par.
* **Upstream:** Ye term tab use hota hai jab aap kisi aur ke project ko **Fork** karte ho (copy karte ho). Us asli owner ke repo ko hum upstream kehte hain taaki hum wahan se updates le sakein.

**FETCH, PULL, CLONE, FORK**

**1. Git Fetch vs Git Pull: Kya farak hai?**

Zadatar log isme confuse hote hain, par asaliyat mein Pull do commands ka combo hota hai.

**Git Fetch (Sirf Dekhna)**

git fetch ka matlab hai: *"GitHub par jo bhi naya kaam hua hai, uski jaankari mere computer par le aao, par mere asli code ko mat chhedo."*

* Ye sirf remote repository se data download karta hai.
* Aapka local code waisa hi rehta hai jaisa tha.
* Isse aap check kar sakte ho ki aapke doston ne kya-kya changes kiye hain bina apna kaam disturb kiye.

**Git Pull (Download + Merge)**

git pull ka matlab hai: git fetch + git merge.

* Ye remote se naya code download bhi karta hai aur turant use aapke local code ke saath **Jod (Merge)** deta hai.
* Agar aapne aur aapke dost ne ek hi line change ki hogi, toh pull karte hi **Merge Conflict** aa sakta hai.

**Comparison Tip:** Fetch safer hai kyunki ye aapka code change nahi karta. Pull tab use karo jab aapko pata ho ki sab sahi hai aur aapko naya code apne mein merge karna hi hai.

**2. GitHub mein Forking kya hai?**

**Fork** ka simple matlab hai: Kisi aur ke project ki **"Ek Copy"** apne GitHub account mein bana lena.

* Maan lo koi bada Open Source project hai. Aap uske asli code (Main Repo) mein seedha badlav nahi kar sakte kyunki aapke paas permission nahi hai.
* Jab aap **Fork** button dabate ho, toh us project ki puri copy aapke apne GitHub account par aa jati hai.
* Ab ye aapka "Raj" hai! Aap isme kuch bhi change karo, asli project par koi asar nahi padega.

**3. Contribute Kaise Karein? (The Contribution Flow)**

Open source ya team mein contribute karne ka ek standard tarika hota hai:

1. **Fork:** Sabse pehle asli project ko apne account mein Fork karo.
2. **Clone:** Us Forked repo ko apne computer par download karo (git clone).
3. **Branch:** Ek nayi branch banao (git switch -c my-feature).
4. **Work & Commit:** Changes karo aur git commit karo.
5. **Push:** Apne GitHub account (Forked Repo) par changes bhej do.
6. **Pull Request (PR):** Ye sabse important step hai. Aap asli owner ko ek "Request" bhejte ho: *"Maine aapke project mein ye fix kiya hai, kya aap ise asli project mein merge karoge?"*

**4. Remote: Origin vs Upstream**

Jab aap Fork karte ho, toh do tarah ke "Remote" hote hain:

* **Origin:** Ye aapka apna GitHub wala repo hai (Jahan se aapne clone kiya).
* **Upstream:** Ye woh asli (Original) project hai jahan se aapne fork kiya tha.

Aap upstream se naya code fetch karte rehte ho taaki aapka project purana na ho jaye.

**Summary Table:**

| **Term** | **Kya hai?** | **Kyun use karein?** |
| --- | --- | --- |
| **Fetch** | Download information | Sirf dekhne ke liye ki remote par kya naya hai. |
| **Pull** | Download + Merge | Naya code apne local mein turant update karne ke liye. |
| **Fork** | Personal Copy | Kisi aur ke project par kaam karne ke liye (binna permission ke). |
| **PR** | Request to merge | Apne kaam ko asli project mein shamil karwane ke liye. |