**GIT(Source Version Control System)**

Permbajtja

[Hyrje 2](#_Toc62464448)

[Menaxhimi i punës ekipore 2](#_Toc62464449)

[Çka janë VCS (Version Control Systems)? 3](#_Toc62464450)

[Modelet e VCS dhe krahasimi 3](#_Toc62464451)

[GIT 3](#_Toc62464452)

[Çka është Git? 4](#_Toc62464453)

[Veçoritë e Git 4](#_Toc62464454)

[Gjendjet e fajllit 4](#_Toc62464455)

[Tri pjesët kryesore të Git 4](#_Toc62464456)

[Terminologjia 5](#_Toc62464457)

[Rregullimi I hapesires punuese 5](#_Toc62464458)

[Shkarkimi dhe instalimi 5](#_Toc62464459)

[Konfigurimi 5](#_Toc62464460)

[Konfigurimi në nivel të përdoruesit 6](#_Toc62464461)

[Komanda bazike gjatë punës me Git 6](#_Toc62464462)

[Git status 6](#_Toc62464463)

[Git diff 7](#_Toc62464464)

[Git add 7](#_Toc62464465)

[Git commit 8](#_Toc62464466)

[Puna me repository locale 9](#_Toc62464467)

[Krijimi i një repository lokale dhe shtimi i fajllave 9](#_Toc62464468)

[Shikimi i historisë dhe dallimet 11](#_Toc62464469)

[Vendosja e fajllave në staging 11](#_Toc62464470)

[Fshirja dhe riemërimi i fajllave 13](#_Toc62464471)

[Rikthimi i fajllave në gjendjen e mëparshme 13](#_Toc62464472)

[Kthimi në commit të mëhershëm dhe modifikimi i tij 13](#_Toc62464473)

[Pastrimi i kopjes së punës 13](#_Toc62464474)

[Injorimi i fajllave me .gitignore 13](#_Toc62464475)

[Puna me remote repository 14](#_Toc62464476)

[Klonimi i një repo 14](#_Toc62464477)

[Listimi i commits 14](#_Toc62464478)

[Listimi i branches dhe tags 14](#_Toc62464479)

[Marrja e ndryshimeve prej remote 14](#_Toc62464480)

[Marrja dhe bashkimi i ndryshimeve prej remote 14](#_Toc62464481)

[Dërgimi i të dhënave në remote branch 14](#_Toc62464482)

[Krijimi dhe dërgimi i tags në repo 15](#_Toc62464483)

[Puna me branches, rebase dhe cherry-pick 15](#_Toc62464484)

[Krijimi i branch-ave lokale 15](#_Toc62464485)

[Riemërimi dhe fshirja e branch-ave 15](#_Toc62464486)

[Stash 15](#_Toc62464487)

[Bashkimi i branch-ave(merge) 16](#_Toc62464488)

[Rebase 16](#_Toc62464489)

[Cherry-pick 16](#_Toc62464490)

[Krijimi dhe fshirja e një remote branch 16](#_Toc62464491)

[Rast Studimi 17](#_Toc62464492)

[Zbatimi i një projekti grupor në GitHub 17](#_Toc62464493)

[Puna sipas parimit të GitHub 17](#_Toc62464494)

[Krijimi i repository 17](#_Toc62464495)

[Definimi i branch-eve 17](#_Toc62464496)

[Definimi i qasjeve 17](#_Toc62464497)

[Caktimi dhe përcjellja e detyrave 17](#_Toc62464498)

[Pull requests 17](#_Toc62464499)

[Fork 18](#_Toc62464500)

# Hyrje

## Menaxhimi i punës ekipore

Zhvillimi i softuerëve apo çfarë do pune tjetër në sektorin e IT që ka të bëj me fajlla, kërkon që puna që bëhet të jetë e menaxhueshme, duke kontrolluar historinë e punës, perkatësisht fajllave që përbëjnë projektet në zhvillim. Kompleksiteti i menaxhimit të punës në ekip rritet në varësi të madhësisë së ekipit, çfarë ka shtyer që nevoja për një vegël që e kryen këtë punë të jetë më se e nevojshme.

## Çka janë VCS (Version Control Systems)?

Një sistem për kontroll të verzioneve(VCS) është një vegël e cila menaxhon ndryshimet e fajllave dhe direktoriumeve brenda një projekti. Menaxhimi i fajllave nuk është i limituar për një lloj të caktuar të fajllave, andaj mund të përdoren fajlla të natyrave të ndryshme.

Përdorimi i VCS-ve është bërë i domosdoshëm në industrinë e inxhinierisë kompjuterike për shkak të cilësive që zotëron dhe lehtësisë së menaxhimit të një procesi të komplikuar. Disa prej cilësive të një VCS-je janë:

* Çfarë do që ruhet nuk humbet, duke mbetur si histori, andaj gjithmonë mund të kthehemi në versione të mëhershme të fajllit apo projektit për çfarë do arsye që na nevojitet
* Gjatë krahasimit të versioneve të fajllave, mund të shohim se çfarë është ndryshuar mes versionit paraprak dhe versionit aktual, kur është bërë ndryshimi dhe kush e ka bërë atë. Përdoruesi po ashtu njoftohet automatikisht për konfliktet që mund të kenë dy versione, prandaj është shumë e vështirë që të mbishkruajmë fajlla, por gjithsesi është e mundur.
* VCS-të mund të sinkronizojnë punën e përfunduar nga përdorues të ndryshëm në kompjuterë të ndryshëm.

## Modelet e VCS dhe krahasimi

– Ekzistojnë dy modele të VCS: VCS të centralizuara dhe të shpërndara.

* VCS e centralizuar –Përbëhet nga një server qendror i vetëm që i menaxhon të gjitha versionet e fajllave. Kjo u mundëson që një grup të punoj në të njëjtat fajlla duke qenë në dijeni se në cilën pjesë secili prej anëtarëve të grupit ka kontribuar. Si mangësi e këtij modeli është se të gjitha fajllat ruhet vetëm në një vend, ku në rast defekti, secili prej anëtarëve të ekipit do të kishin nga një version lokal të fajllave që nga hera e fundit që kanë pasur qasje në server.
* VCS e shpërndarë – Secili nga klientët të cilët marrin fajllat nga serveri, marrin po ashtu tërë historinë e versioneve të atyre fajllave, ashtu që në rast të dështimit të serverit, cilido nga klientët mund t’i replikojë të dhënat në server. Git, i cili është prej VCS-ve më të përdorur, paraqet një shembull të këtij modeli, për çka do të flasim në modulin e ardhshëm.

# GIT

## Çka është Git?

– Git është një sistem i shpërndarë për kontroll të versioneve (DVCS) e cila kontrollon ndryshimet e kodit burimor gjatë zhvillimit të softuerit. Git nuk është i limituar në një tip të caktuar të fajllave, por zakonisht përdoret në zhvillime softuerike.

## Veçoritë e Git

– **Git është lokal** – pas marrjes së projektit lokalisht, e tërë historia e projektit gjendet në makinën tonë. Në këtë mënyrë puna jonë mund të vazhdoj edhe pa qasje në serverin ku ruhet projekti dhe të bashkohet me pjesën tjetër kur do që qasja në server është e mundshme.

– **Git ka integritet** ashtu që nuk mund të ndryshohet një fajll, duke mos qenë Git në dijeni. Secilit ndryshim në fajll i gjenerohet një vlerë 40-shifrore hash, ashtu që nëse dy persona në makina të ndryshme shikojnë versionin me hash të njëjtë, ata janë duke shikuar fajllat që kanë të njëjtën përmbajtje.

– Një veçori që e dallon Git nga VCS-të tjerë është mënyra se si ruhen fajllat. Në VCS-të tjerë fajllat ruhen si një listë e ndryshimeve të fajllave, pra në secilin version ruhen vetëm ndryshimet e secilëve prej fajllave, çka e bën të vështirë të shihet statusi i tërë projektit në një version të caktuar. Git në anën tjetër ruan fajllat si **snapshots**(fotografi) të versioneve të të gjithë fajllave brenda projektit. Sa herë që ne ndryshojmë diçka, Git bën një snapshot të të gjithë fajllave brenda projektit duke shikuar versionin e secilit prej tyre. Në rast se një fajll nuk është ndryshuar ai nuk ruhet përsëri, por ruhet një referencë tek versioni i vjetër i atij fajlli, ashtu që për çdo version mund të shohim statusin e plotë të projektit.

## Gjendjet e fajllit

– Secili fajll brenda një git projekti mund të jetë në njërën nga fazat vijuese:

1. **Commited** – fajlli gjendet në bazën e të dhënave lokale të projektit
2. **Tracked** – fajll i cili ka qenë pjesë e snapshot-it të fundit
3. **Unmodified** – fajll i cili nuk  është modifikuar që nga commit-i i fundit, por që do të përfshihet në commit-in e radhës në gjendjen që e ka.
4. **Modified** – fajlli është ndryshuar prej snapshotit që gjendet në bazën e të dhënave lokale dhe do të përfshihet në commit-in e radhës, por me përmbajtjen e re.
5. **Staged** – ndryshimi i fajllit ka përfunduar dhe tani është i gatshëm të kalojë në bazën e të dhënave lokale ne fazën commited duke krijuar snapshot të ri
6. **Untracked** – është një gjendje tjetër të cilën e kalon fajlli vetëm pas krijimit të tij brenda projektit. Këta janë fajllat të cilët nuk kanë qenë pjesë e snapshotit të kaluar, por janë shtuar në versionin aktual. Këta fajlla duhet të shtohen në fazën staging dhe pastaj të bëhen commit në bazën e të dhënave lokale.

## Tri pjesët kryesore të Git

Tri pjesët kryesore të git-it janë:

1. .git direktoriumi – fajllat në këtë gjendje janë bërë commit dhe janë ruajtur si snapshot versione
2. Staging area – fajllat në këtë gjendje janë ndryshuar dhe janë vendosur si të gatshëm për snapshotin e commit-it të ardhshëm
3. Working area – ndryshimet e fajllave që prej checkout-it të fundit që nuk janë shtuar në staging area për commit.

## Terminologjia

**Repository** – një Git repository është një direktorium ku ruhen të gjitha fajllat, direktoriumet dhe përmbajtja që nevojitet për një projekt. Pra, është baza e të dhënave të një projekti që ruan gjithçka prej fajllave de deri te versioni i tyre, fshirjet etj.

**Branch** – është një version i repository që dallon nga projekti kryesor. Është një veçori esenciale në shumicën e VCV-ve modern. Një projekt në Git mund të ketë shumë branches, të cilat përkrahin operacionet e emërimit, ri-emërimit, fshierjes etj.

**Clone** – është kopje e një repository ose fillimi i procesit të kopjimit të një repo. Nëse përdoruesi dëshiron një kopje lokale të një repo në GitHub, git clone lejon krijimin e një kopje lokale të asaj repo në një direktorium lokal prej URL-it të repos.

**Master** – branch-i primar i të gjitha repositories. Të gjitha ndryshimet të bëra commit duhet të jenë në branch-in master. Puna mund të zhvillohet direkt në këtë branch apo të krijohen branch-a specifik.

**Push** – përditëson një remote branch me commits të bërë në branch-in aktual. Pra, ndryshimet po shtyhen prej branchit lokal në atë remote.

**Index** – cache ku ruhen ndryshimet përpara se ato të bëhen commit

**Commit** – ruan përmbajtjen e tanishme në indexin e commit-it të ri së bashkë me një mesazh që shpjegon ndryshimet e bëra nga përdoruesi.

# Rregullimi I hapesires punuese

## Shkarkimi dhe instalimi

Instalimi i Git bëhet përmes linkut <https://git-scm.com/>, ku varësisht nga sistemi operativ shkarkohet fajlli për ekzekutim.

Pas, përfundimit të instalimit të Git-it, verifikimi bëhet duke ekzekutuar komandën **git --version** në command line.

## Konfigurimi

Konfigurimi i Git mund të bëhet në tri nivele.

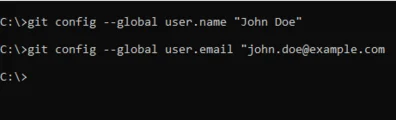
1. Konfigurimi në nivel të sistemit – Bëhet përmes komandës **git config –system** dhe ruhet në direktoriumin */etc/gitconfig* ose *C:/Program Files/Git/etc/gitconfig*
2. Konfigurimi në nivel të përdoruesit – Bëhet përmes komandës **git config –global** dhe ruhet në direktoriumin *~/.gitconfig* ose *C:/Users/<NAME>/.gitconfig*
3. Konfigurimi në nivel të repository – Bëhet përmes komandës **git config** dhe ruhet në direktoriumin *.git/config* brenda secilës repository.

## Konfigurimi në nivel të përdoruesit

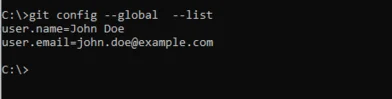
– Pas instalimit të Git disa shembuj për konfigurim mund të jenë konfigurimi i emrit të përdoruesit dhe emailit të tij. Në vazhdim janë komandat që e kryejnë një punë të tillë:

git config –global user.name “John Doe”

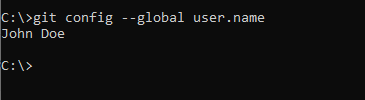
git config –global user.email “[*john.doe@example.com”*](mailto:john.doe@example.com)



– Me komandën **git config –global –list** shihen të gjitha konfigurimet e bëra deri më tani në nivel të përdoruesit. Lista e konfigurimeve të mundshme është e gjatë, por shembujt e mësipërm japin vetëm një hyrje se si mund të konfigurohen preferencat e përdoruesit për Git.



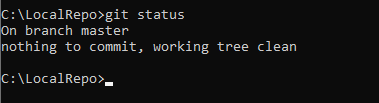
– Në rast se dëshirojmë të kemi vlerën për një konfigurim specifik të bërë atëherë atë mund t’a marrim përmes komandës **git config –global <CONFIG-KEY>**. Pra, nëse dëshirojmë të kemi emrin e përdoruesit i cili gjendet në konfigurim atë e bëjmë përmes komandës **git config –global user.name**, e që në rastin tonë do të shfaqej emri John Doe.



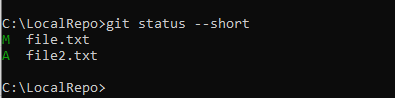
# Komanda bazike gjatë punës me Git

## Git status

Git status  është komandë për kontrollimin e statusit të projektit në çfarë do kohe. Si rezultat na kthehet branchi aktual në të cilën gjendemi, statusi nëse jemi të sinkronizuar me branch-in e njëjte por që gjendet në server si dhe shfaqen fajllat që janë ndryshuar nëse ka, apo mesazhi se nuk ka asgjë për commit.



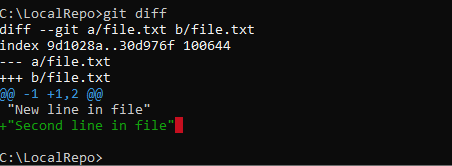
Nëse dëshirojmë që përgjigja e komandës të jetë më e shkurtër atëherë komandës git status i jepet edhe flag-u –short, çka rezulton me shfaqjen e fajllave me nga një shkronjë përpara që tregon statusin e tyre ku M do të thotë Modified(e ndryshuar) dhe A do të thotë Added(e shtuar).



## Git diff

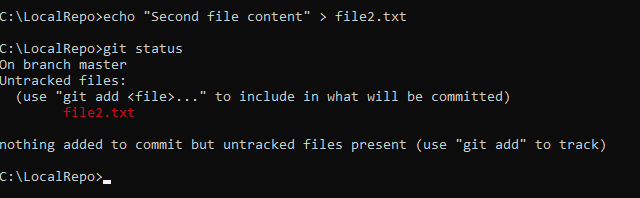
Git diff – informatat që na kthehen pas komandës git status mund të mos jenë të detajuara dhe të mos kenë informacionet të cilat neve na nevojiten. Si plotësim i kësaj komandë është komanda git diff –staged, që krahason ndryshimet e bëra në fajlla. Si përgjigje e kësaj komande shfaqen rezultatet si në vijim:

1. Shfaqen emrat e fajllave të krahasuar
2. Shfaqen metadata të fajllave si SHA1 dhe modi i fajllit
3. Indikatorët e ndryshimit tek fajllat A/B
4. Header – që tregon rreshtin në të cilën fillon ndryshimi, dhe në sa rreshta pas tij ka përmbajtje të ndryshme.
5. Përmbajtja e ndryshuar

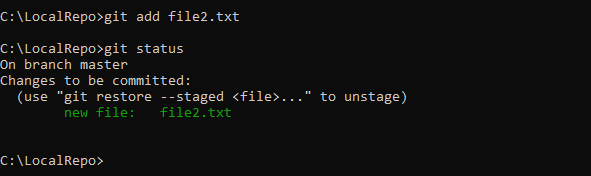


## Git add

Git add – është komandë e cila shton fajllat e krijuar në snapshotin aktual të cilët nuk kanë qenë pjesë e snapshotit në server. Pas krjimit të një fajlli të ri status i tij është untracked.



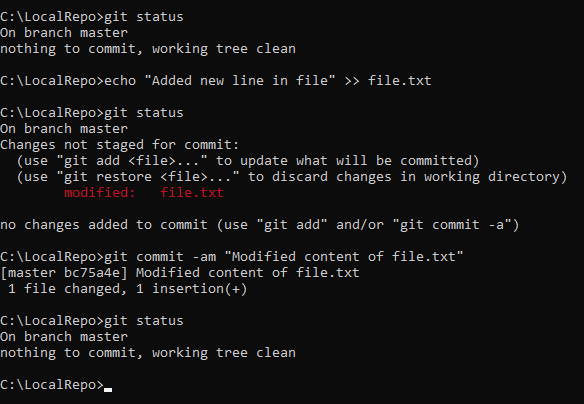
Pas ekzekutimit të komandës git add <FILENAME>, fajlli i ri bëhet pjesë e git projektit duke  ndërruar statusin prej untracked në tracked dhe duke u vendosur në staging area.



## Git commit

Git commit – është komandë e cila ruan ndryshimet në fajlla në snapshotin e radhës. Në rast se gjatë ekzekutimit të komandës git commit kemi fajlla të cilët janë shtuar rishtas, ata nuk do të bëhen pjese e commit pa u ndërruar statusi i tyre nga untracked në tracked. Sa i përket fajllave të modifikuar, fillimisht duhet të shtohen në staging area e pastaj të bëhen commit, mirpo ky proces mund të kryhet me një komandë të vetme duke i shtuar Komandë git commit flagun -a, çka i bashkon të gjitha fajllat e krijuar në commitin e ardhshëm.

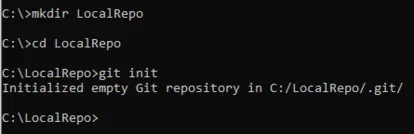
Pra, me komandën **git commit -a -m “Modified content of file.txt”**, kemi shtuar fajllat e modifikuar nw staging area si i dhe kemi ruajtur me një mesazh ‘shtimi i fajllave’



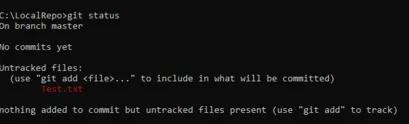
# Puna me repository locale

## Krijimi i një repository lokale dhe shtimi i fajllave

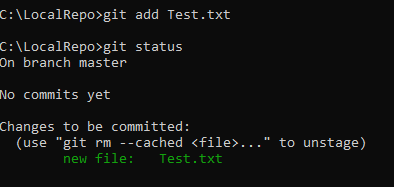
Krijimi i një repo, siç edhe është cekur në modulet e mëhershme bëhet përmes komandës **git init**, pasi që kemi kaluar në direktoriumin e dëshiruar për krijim të saj. Ekzekutimi i kësaj komande rezulton me mesazhin se kemi inicializuar një repository të zbrazët.



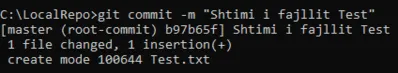
Pas kësaj vazhdojmë me shtimin e një fajlli në repo duke e pasuar me ekzekutimin e komandës **git status**, e cila na tregon se kemi një fajll i cili ka statusin untracked.



Pas komandës **git add <FILENAME>**, statusi i fajllit ndërron prej untracked në fajll të ri(neë file).



Me komandën **git commit -m “shtimi i fajllit test”**, kemi ruajtur fajllin e krijuar në repo-n lokale të cilën e kemi krijuar.



Me komandën **git log**, mund të shohim historinë e ruajtës së fajllave që kemi bërë(commits) në repon tonë.



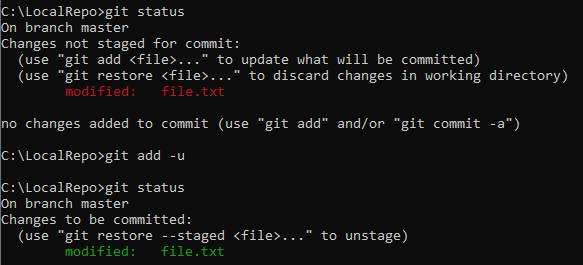
## Shikimi i historisë dhe dallimet

Pasi që të kemi më shumë se një commit në repo dhe dëshirojmë të shohim ndryshimet që kanë ndodhur prej njërit në tjetrin atëherë mund të ekzekutojmë komandën **git diff <First commit SHA1>…<Second commit SHA1>**.

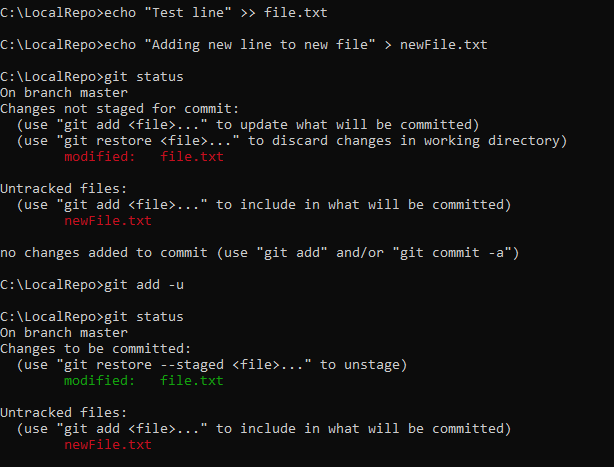
Meqenëse puna me hash është më e vështirë, kjo mund të lehtësohet me përdorimin e **HEAD**, çka njihet si commit-i i fundit i bërë në repo. Me përdorimin e tilde(~) mund të shkojmë prapa në commits, pra **HEAD~1** na dërgon një commit më herët se i fundit, andaj rezultati i njëjtë mund të arrihet duke ekzekutuar komandën **git diff HEAD~1..HEAD**. Në rast se komandës git diff i jepet vetëm një commit, atëherë atë e krahason me commitin e fundit pra HEAD, andaj komanda jonë mund të shkurtohet edhe më tutje duke dhënë rezultatin e njëjtë në **git diff HEAD~1..**.

## Vendosja e fajllave në staging

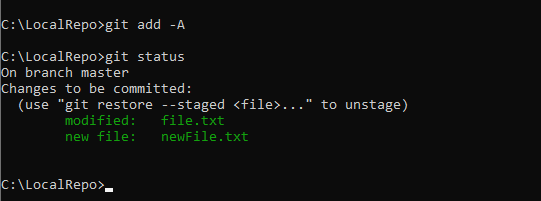
Pas përfundimit të punës në repo, hapi i radhës është kalimi i fajllave në staging, për të qenë gati për commit. Komanda që e bën këtë është **git add -u**, e cila ndërron statusin e të gjithë fajllave të modifikuar si staged.



Mirëpo, duhet pasur kujdes kur kemi fajlla të ri të cilët e kanë statusin untracked, pasi këta fajlla nuk përfshihen nga komanda e mësipërme.



Me qëllim të vendosjes së untracked fajllave në staging, duhet të përdoret komanda**git add <FILENAME>** ose nëse dëshirojmë që t’i shtojmë të gjithë fajllat atëherë komandën **git add -A**. Me këtë rast fajllat e rinj janë të gatshëm për commit. Megjithatë duhet pasur kujdes gjatë përdorimit të kësaj komande pasi mund të shtohen edhe fajlla të cilët mund të jenë të pa dëshiruar.



Fajllat e modifikuar mund të vendosen në hapsirën staging si të ndara, duke mundësuar edhe bërjen commit të tyre si të ndarë. Pra, në rast se kemi dy fajlla të modifikuar, mund të shtojmë të parin si staged, pastaj ta bëjmë commit, dhe të njëjtën procedurë ta përsërisim edhe për fajllin e dytë.

## Fshirja dhe riemërimi i fajllave

Pas fshirjes së një fajlli brenda repos, duhet që operacioni i fshirjes të shtohet në staging area dhe të bëhet commit me qëllim të largimit të fajllit nga snapshot.

Në anën tjëtër pas riemërimit të një fajlli brenda repos, nëse e shikojmë statusin e vërejmë se fajlli me emrin e vjetër është i markuar si i fshirë, ndërsa fajlli i njëjtë por me emër të ri njihet si një fajll i ri i shtuar. Pas shtimit të ndryshimeve në staging area, git kontrollon përmbajtjen e fajllave dhe vëren se operacioni i kryer ka qenë vetëm riemërim dhe është i gatshëm që të bart ndryshimin në snapshotin e ardhshëm.

## Rikthimi i fajllave në gjendjen e mëparshme

Në rast modifikimit të një fajlli në mënyrë të gabueshme apo për çfarë do arsye tjetër ekziston mundësia që fajlli të rikthehen në versionin e fundit që gjendet në repo. Kjo arrihet duke ekzekutuar komandën **git checkout <FILENAME>**.

Kur në repo ka më shumë se një fajll të modifikuar, komanda e mësipërme mund të jetë më e lodhshme për tu kthyer në versionin fillestar. Për këto raste, na vije në ndihmë komanda **git reset –hard**, e cila i kthen të gjitha ndryshimet në të gjitha fajllat mrapa.

## Injorimi i fajllave me .gitignore

Gjatë punës mund të kemi raste kur kemi fajlla apo direktoriume të cilët ndryshojnë gjatë gjithë kohës. Një shembull i tillë mund të jetë ndonjë fajll konfigurues i editorit që përdoret apo ndonjë direktorium ku ruhen logs të programit. Me qëllim që këta fajlla të mos bëhen commit, ata mund të shtohen në fajllin **.gitignore** që të injorohen gjatë këtij procesi.

Shtimi i fajllave apo direktoriumeve mund të bëhet në mënyrë absolute apo relative varësisht nga dëshira e përdoruesit. Sa i përket fajllave, mund të përfshihet fajllat me prapashtesën e caktuar si p.sh \*.txt, të gjithë fajllat me prapashtesën txt.

EX. Krijojme fajllin .gitignore me komanden: *touch .gitignore*

C’do fajll ose folder qe shtohet ne fajllin .gitignore do injorohet nga GIT edhe nese bejme ndryshime ne te.

# Puna me remote repository

## Klonimi i një repo

Klonimi i një repo bëhet përmes komandës **git clone <REPO\_URL>**, i cili si proces përfundon relativisht shpejtë pa marr parasysh madhësinë e repos. Me klonim përdoruesi fiton një kopje të repos në makinën lokale, me të gjithë historinë e commits. Historia e commits mund të shikohet përmes komandës git log, apo në mënyrë më të pastër përmes komandë **git log –oneline**.

## Listimi i commits

Listimi i commits bëhet me komandën **git log**, siç edhe është cekur më lartë. Në rast se dëshirojmë të shohim commit-in e fundit atëherë këtë e bëjmë përmes komandës **git show HEAD**. Shikimi i ndryshimeve në commit të caktuar mund të bëhet edhe duke përdorur SHA1 kodin e commit, pra **git show <COMMIT\_SHA1>**.

## Listimi i branches dhe tags

Branches përdoren si kopje të përkohshëm të punës me qëllim që të ndahet linja kryesore e zhvillimit na përmirësimi i gabimeve. Listimi i tyre bëhet me komandën **git branch**, ndërsa listimi i remote branches bëhet me komandën **git branch -r.**

Tags në anën tjetër paraqesin pika stabile të kodit. Pra, vendosen etiketa në historinë e repos si pika të rëndësishme gjatë zhvillimit. Listimi i tyre bëhet me komandën **git tag**.

## Marrja e ndryshimeve prej remote

Para se të marrim ndryshimet prej remote duhet të kontrollojmë nëse jemi të lidhur me një remote repo. Kjo bëhet përmes komandës **git remote -v.** Lidhja me remote repo bëhet në mënyrë automatike kur ndodh klonimi.

Nëse repo jonë është lokale dhe dëshirojmë të caktojmë një destinacion remote atë e bëjmë përmes komandës **git remote add <REMOTE\_NAME> <REMOTE\_REPO\_URL>**.

Pasi që të jemi lidhur me remote repo mund të ekzekutojmë komandën **git fetch**, e cila i merr ndryshimet nga remote repo. Në rast se gjatë inspektimit të commits me komandën git log lokalisht dhe remote ka ndryshime, atëhere duhet të bëhet bashkimi i remote branch me branchin lokal me komandën **git merge <REMOTE\_NAME>/<BRANCH\_NAME>**.

## Marrja dhe bashkimi i ndryshimeve prej remote

Procesi i marrjes së ndryshimeve përmes komandës fetch  dhe pastaj me komandën merge mund të zëvendësohet me një komandë të vetme e cila është **git pull**. Para se të ekzekutohet kjo komandë duhet që branchit lokal t’i caktohet një remote branch, të cilin branchi lokal e pasqyron. Kjo arrihet përmes komandës **git branch –set-upstream <LOCAL\_BRANCH\_NAME> <REMOTE\_NAME>/<REMOTE\_BRANCH\_NAME>**.

## Dërgimi i të dhënave në remote branch

Pas modifikimit të fajllave në branchin lokal, si dhe commitit të tyre, duhet të vazhdojmë me hapin e radhës, i cili është dërgimi i të dhënave të ndryshuara në remote branch. Kjo bëhet përmes komandës **git push**, e cila e dërgon commitin lokal në remote branch. Duhet pasur kujdes, pasi varësisht nga protokolli që përdorni gjatë caktimit të remote repo, pas komandës git push mund të keni nevoje ti shkruani kredencialet e juaja për qasje në repo.

Top of Form

Bottom of Form

# Puna me branches

Top of Form

## Krijimi i branch-ave lokale

Krijimi i një branch-i bëhet përmes komandës **git branch <BRANCH\_NAME>**. Pas krijimit  të branchit duhet të kalojmë tek ai përmes komandës **git checkout <BRANCH\_NAME>**. Këto dy komanda mund të realizohen përmes një komande të vetme: **git checkout -b <BRANCH\_NAME>**, ku krijohet branchi dhe kalohet menjëhere në të pa pasur nevoje për komanda shtesë.

Krijmi i branch-ave mund të bëhet nga një commit i caktuar përmes komandës **git branch <BRANCH\_NAME> <COMMIT\_SHA1>**.

Dallimi mes branch dhe tag është se branch do t’i ndjek commits, duke i shtuar ato brenda vetes, përderisa tag do të qëndroj gjithmonë në të njëjtin commit, pra është vetëm një emër më i lehtë për një SHA1 kod.

## Riemërimi dhe fshirja e branch-ave

Riemërimi i branch-ave bëhet përmes komandës **git branch -m  <OLD\_NAME> <NEW\_NAME>**.

Fshirja e një branch-i mund të bëhet përmes komandës **git branch -d <BRANCH\_NAME>**, e cila mund të rezultojë si jo e suksesshme duke pasur vërejtje se nuk është bashkuar në master apo ndonjë mesazh tjetër. Nëse përdoruesi dëshiron që përkundër vërejtjes t’a fshij branchin në fjalë, kjo realizohet përmes komandës **git branch -D <BRANCH\_NAME>**.

## Stash

Gjatë punës në një branch ndodh që të kemi nevojë të kalojmë në një branch tjetër për të përfunduar një punë të caktuar. Kalimi prej branchit në branch tjetër përmes komandës **git checkout <BRANCH\_NAME>** nuk është i lejuar pa i bërë commit fajllat në branchin aktual. Me qëllim të ruajtës së punës në branchin aktual na vije në ndihmë komanda **git stash**, e cila ruan një kopje të punës aktuale në memorie.

Pas ekzekutimit të kësaj komande jemi të lirë të kalojmë në branchin që dëshirojmë dhe kur të rikthehemi në branchin e tanishëm mund të e rikthejmë edhe punën e bërë përmes komandës **git stash apply**.

Me komandën **git stash list** mund të shohim se çka përmban stash lista, ku pas komandës apply, ndryshimet gjenden ende brenda listës.

Me qëllim të aplikimit të stash në branch dhe fshirjes së saj nga lista mund të ekzekutojmë komandën **git stash pop**.

Largimi i një elementi nga stash lista mund të bëhet edhe me **git stash drop** e cila e largon elementin e fundit të futur në listë.

Në raste të caktuara mund edhe të krijojmë branch të ri prej elementit të stash listës me komandën **git stash branch <BRANCH\_NAME>**, ku krijohet branchi me emrin e caktuar dhe kalohet direkt aty.

## Bashkimi i branch-ave(merge)

Bashkimi i dy branchave bëhet përmes komandës **git merge <BRANCH\_NAME>,** e cila bashkon ndryshimet e branchit të specifikuar në branchin aktual. Në disa raste ndodh që të jetë e modifikuar e njëjtja pjesë e fajllit në të dy branch-at, dhe për këtë arsye shfaqen konflikte, të cilat duhet të rregullohen përmes mergetool të caktuar nga përdoruesi. Varëisisht nga zgjedhja e tij mund të merret ndryshimi nga një branch i caktuar apo të bashkohen ndryshimet e te dy brancha-ve në mënyrë që të vazhdoj procesi i bashkimit.

## Krijimi dhe fshirja e një remote branch

Në rast të bashkimi të një branch-i lokal me master, ai nuk do të jetë pjesë e remote repo si branch i veçantë. Nëse dëshirojmë që një branch të cilin e kemi lokalisht ta krijojmë edhe remote, këtë e arrijmë përmes koandës **git push origin <BRANCH\_NAME>**.

Fshirja e një branchi remote bëhet duke ekzekutuar komandën **git push origin :<BRANCH\_NAME>**

# **Rast Studimi**

## Zbatimi i një projekti grupor në GitHub

GitHub është një shërbim i cili ofron kontrollim të versioneve për kodin burimor të programeve të ndryshme. Është një prej shërbimeve më të përdorura, nëse jo më i përdoruri në këtë lëmi, andaj do të përdoret edhe nga në për një projekt grupor.

## Puna sipas parimit të GitHub

Puna në një repo në GitHub bëhet në bazë të disa rregullave. Fillimisht përdoruesi duhet të krijojë një branch në repo. Pas krijimit të branchit është i lirë që të krijoj, fshij dhe modifikoj fajllat brenda branch-it të tij.

Si hap vijues, pasi që përdoruesi vlerëson se puna ka përfunduar, dërgon një pull request tek administratori/ët e repos, nga fillon diskutimi për alternativat e mundshme të punës së bërë, vërejtjeve apo gabimeve që mund të ketë.

Varësisht nga sugjerimet që merr, përdoruesi modifikon branchin dhe vazhdon bashkimin e pjesës së kodit, çka e përditëson në mënyrë automatike edhe pull requestin.

Në rast se puna është vlerësohet si e suksesshme nga mbikqyrësi, pjesa e punuar bashkohet me branchin që ka si destinacion, kurse branchi aktual fshihet nga repository.

## Krijimi i repository

Krijimi i repository bëhet përmes faqes pasi që përdoruesi të logohet, ku zgjidhet emri i repos, pastaj nëse duhet të jetë private apo publike etj.

Pas krijimit si repo e zbrazet ofrohen opsionet e inicializimit, përmes command line si repo e re, nga një repo ekzistues apo importimi i kodit nga një repo tjetër.

## Definimi i branch-eve

Pas krijimit te repos, duhet të caktohet branch-i kryesor, si dhe branch-at tjerë të nevojshëm për zhvillim. Pas krijimit të branch-ave vendosen rregullat për mbrojtje të tyre se kush mund të bëj push kodin dhe kush mund të bashkoj kodin në ata branch-a.

## Definimi i qasjeve

Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.

## Caktimi dhe përcjellja e detyrave

Detyrat mund të krijohen si zhvillime të reja brenda projektit, duke iu alokuar një apo më shumë personave. Detyrat mund të përdoren edhe për adresimin e çështjeve tjera, si dëshirë për një shtim të një veçorie tjetër, raportim i gabimeve si dhe të përcillen deri në zgjidhje apo përfundim të tyre.

## Pull requests

Pas punës në një branch, krijohet një pull request për të treguar se puna e caktuar është përfunduar. Gjatë krijimit të pull requestit mund të shihet ndryshimet që kanë ndodhur nga versioni paraprak. Bashkëpunëtorët tjerë mund të propozojnë zgjidhje tjera, apo të komentojnë mbi zgjidhjen e ofruar. Në rastin kur të gjitha komentet zgjidhen, personi i caktuar si mbikëqyrës i pull requestit e mbyll atë, duke e bashkuar me pjesën tjetër të kodit. Në raste të caktuara pull requesti mund të lidhet me një detyrë, për të treguar punën në progres për të.

## Fork

Fork është komandë e cila lejon kopjimin e një repo publike, duke ofruar mundësine e eksperimentimit me repon e kopjuar pa ndikim në repon origjinale.

Top of Form

Top of Form

Top of Form

Top of Form

Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Pas krijimt të repos, mund të ftohen bashkëpunëtorë tjerë që të bëhen pjesë e repos. Varësisht nga lloji i llogarisë që keni në GitHub mund të caktoni edhe nivelin e qasjes së përdoruesëve të tjerë. Për llogari përsonale ekzistojnë vetëm dy role: repository owner dhe collaborator, ku secili prej tyre ka nivelin e caktuar të qasjes.Top of Form

Top of Form