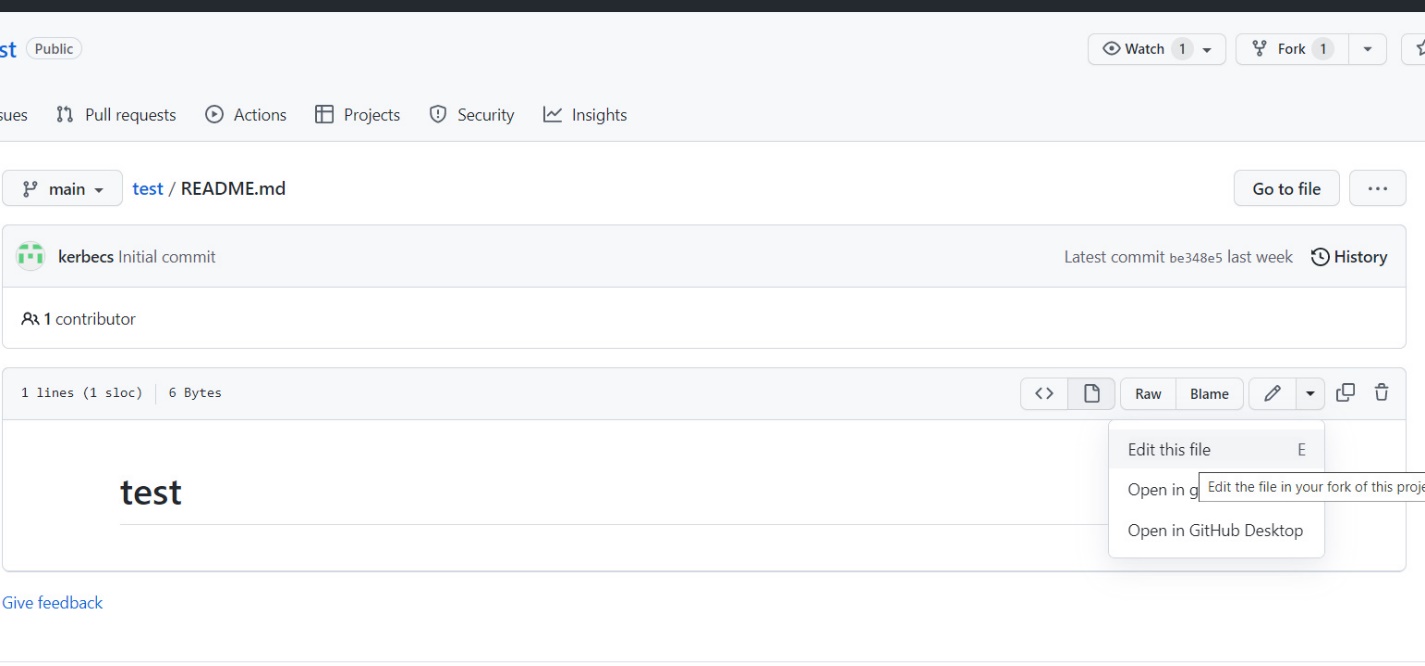
**Pull Request**

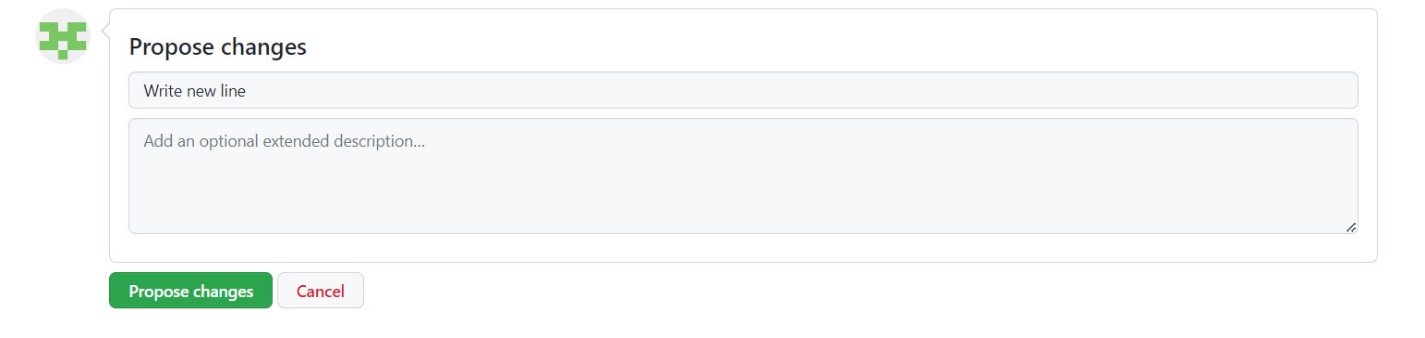
* Odata ce facem un proiect public pe GitHub, oricine il poate vedea, downloada si face pull request
* **Fork(ing)** – un mod de a crea o copie a unui anumit repozitoriu in asa fel incat sa ne apartina. Adica, userii vor putea da push la forked copy, care le apartin lor, dar sunt doar copii ale proiectului, si nu modifica propriu zis proiectul original.
* **Cand lucram la proiecte hostate pe github, trebuie intai sa cream intai neaparat un fork(copy) a repozitoriului original si sa lucram pe acea copie.**
* Un forked repo este ca un normal repo, doar ca GitHub stie de la care repository s-a creat forked repo si putem sa trimitem modificarile noastre de la forked repo in original repo creand un **pull request.**
* **Pull Request** – commit sau serie de commituri create de noi pe care le trimitem la ownerul la repo original pentru ca ele sa fie adaugate in el.
* nu toti au acces la original repo, ci doar unele persoane autorizate. Oricine poate pe GitHub sa trimita pull request. Ownerul le va putea vedea si decide daca sa le pune sau nu in repo original

**Pasi de urmat**

* Ne ducem la repsitory a cuiva si selectam un fisier, si dam edit file:

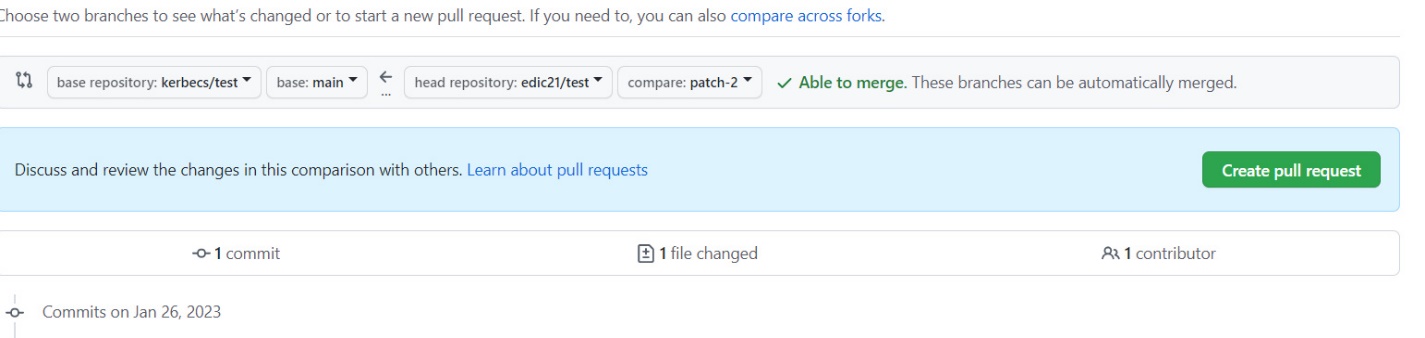


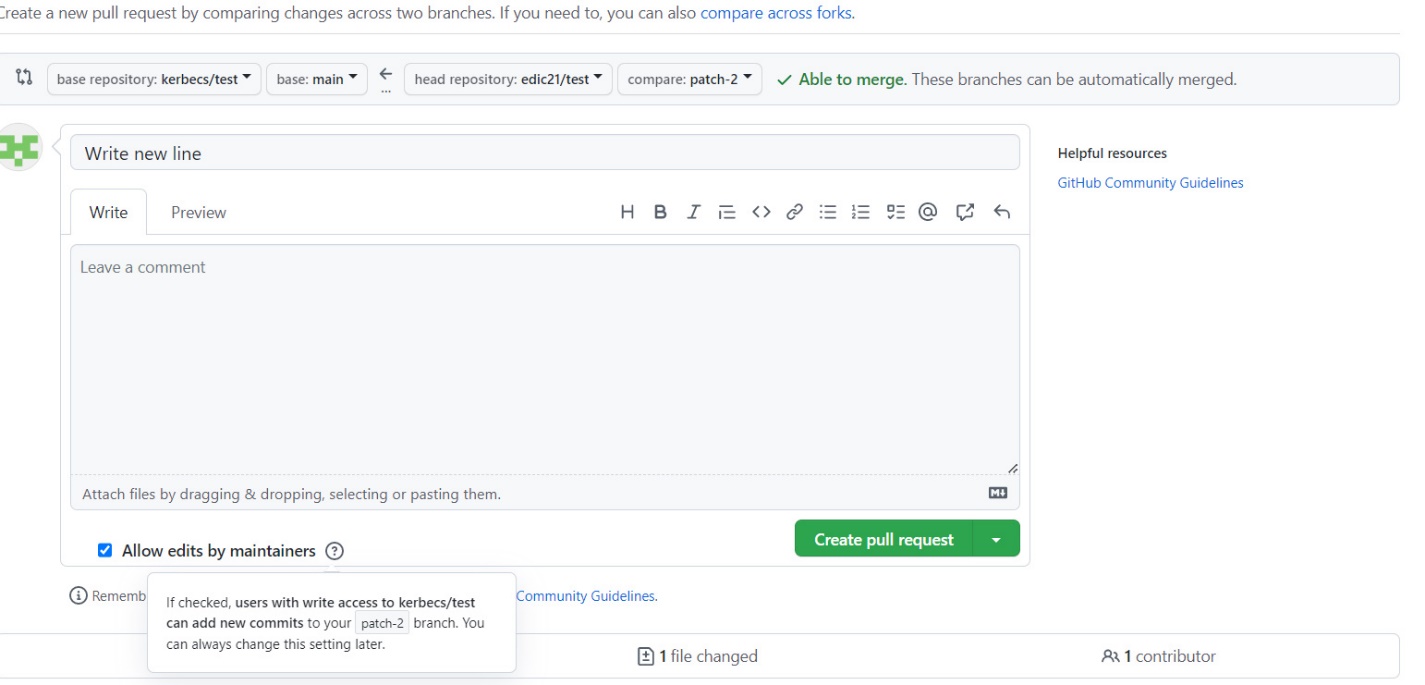
* Modificam ce avem de modificat, si dam Propose changes



Propose changes nu trimite inca pull requestul, ci il creaza pentru noi deocamdata.

* Cand totul e gata, dam create pull request



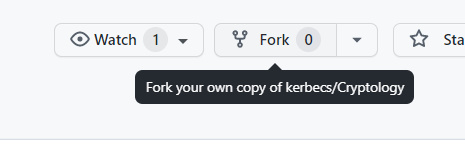
* 

Allow edits my maintainers – dam acces la ownerii commitului sa modifice branch a noastra in care se afla modificarile facute.

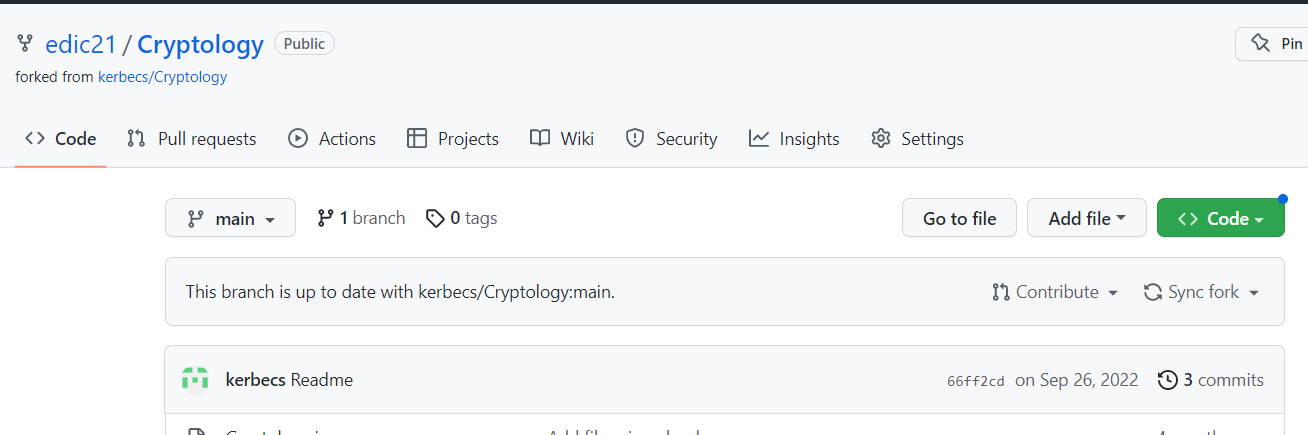
Vedem ca acest commit se afla in patch-2, care e branch a noastra, deci deocamdata modificarile sunt doar la noi.

**Copy the repo**

Putem copia chiar tot repo. Pentru asta, apasam pe Fork, si vom crea o copie.



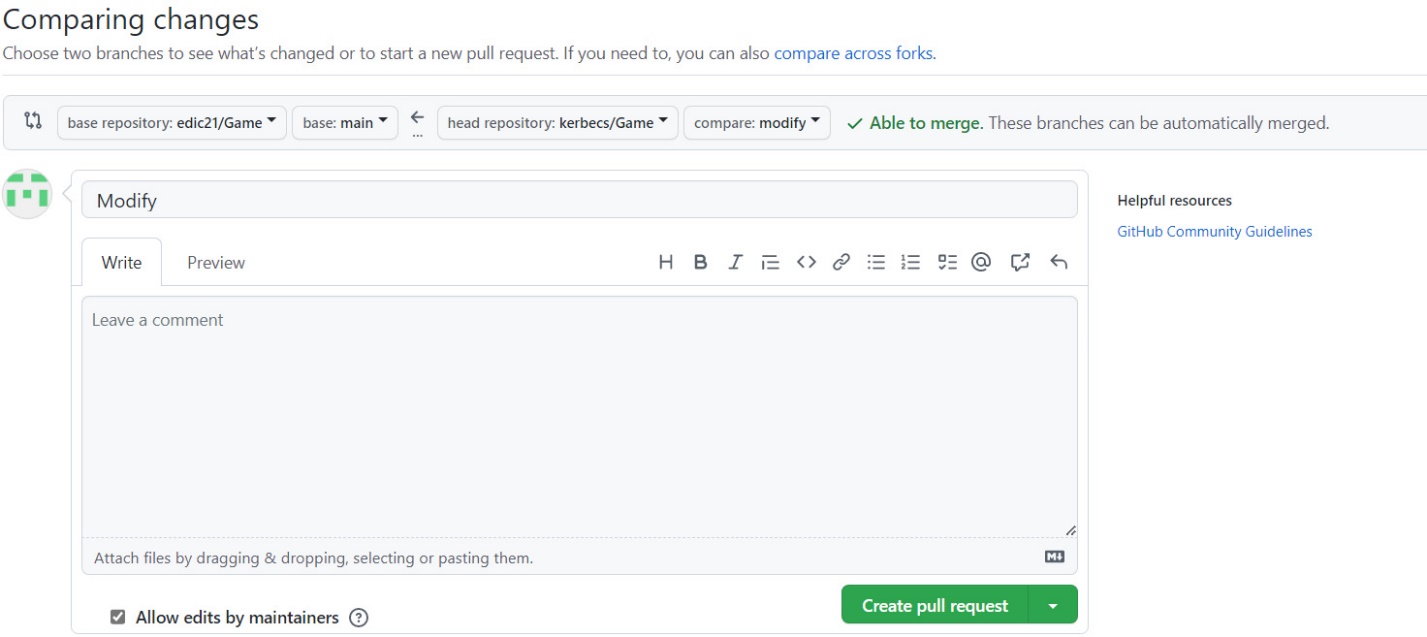
Repo va fi creat ca al nostru, dar va scri ca nu ne apartine:





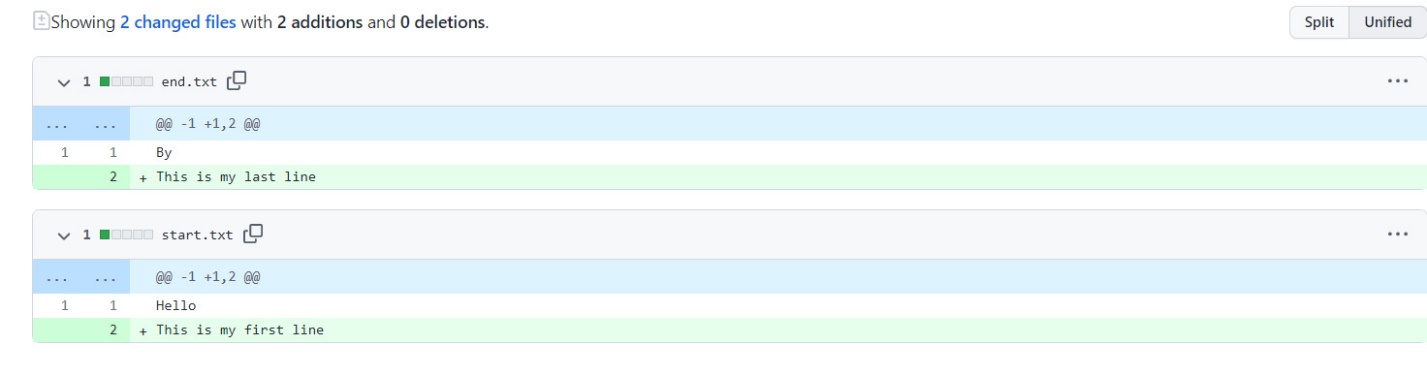
**Pull request in command line**

* Pull request de pe github e bun doar ca sa facem modificari simple si scurte, insa nu e prea bun cand vrem sa modificam mai multe fisiere.
* Iata de ce, trebuie sa copiem repo la noi in profil, adica facem un folk, si deci dam git clone in pc la acel repo floked(copiat la noi)
* Putem vedea toata istoria commitului facuta de owner cu git log, deci toate informatiile ne sunt disponibile.
* Cel mai bine e sa cream o noua branch, sa modificam ce avem in ea, ca sa pastram in main proiectul original, si sa o incarcam cu **git push -u origin branch** in floked remote repo si sa o trimitem ca sa fie merge cu main din repo la owner.
* Apoi deja intram pe github si dam pull request la noul branch
* E forte important sa verificam, cand dam pull request, daca branch poate fi supusa unui merge la main branch a ownerului cu succes:



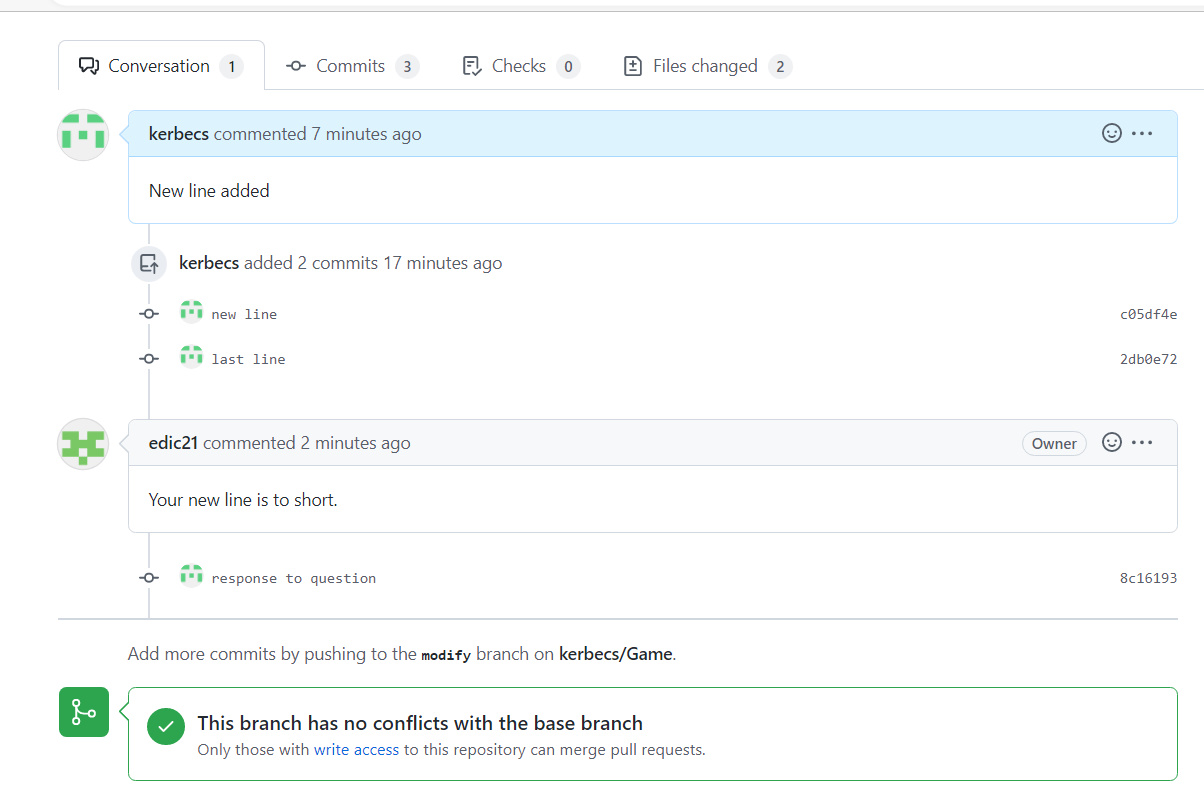


* Jos se face un diff intre main branch a ownerului, care vrem sa dea merge la acest branch creat de noi



**Updating an existing pull request**

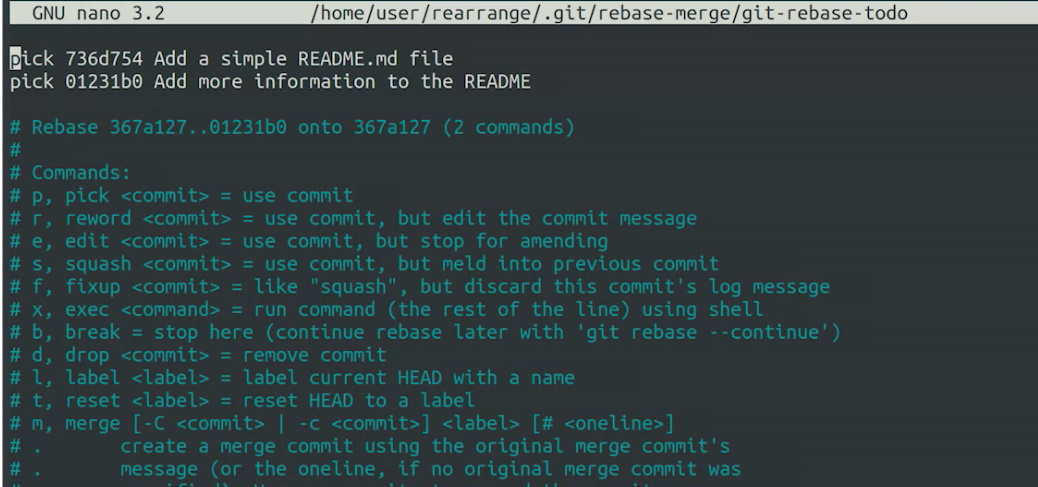
* Dupa ce am trimis un pull request, posibil sa primim vreo cerere de la owner la repo ca sa mai facem ceva in plus.
* Deci, modificam ce avem de modificat si dam iar git push, dar acum vedem ca fara a face vreun pull request, git a facut automat unul cu ceea ce am trimis noi cu git push, si este ca raspuns la cerere. Doar prima data e necesar sa dam request manual la un nou branch, mai departe totul se va face automat :



* Daca totusi vrem sa cream un separate pull request, trebuie creat un nou branch

**git rebase -i**

* -i – interactiv
* aceasta este interactiv rebase, si ofera actiuni ce se vor face dupa ce commitul din branch ales se va combina cu fiecare, sau putem doar sa editam commiturile din branch curent, fara vreun combine, cu git rebase -i HEAD~





commiturile din branch in care scriem commanda pana la pozitia unde ar trebui sa stea noul commit pus

**commit care vine din branch scris in comanda si trebuie sa se combine cu restul din branch curent(de mai sus de el)**

* Cand vom scrie asta, se va deschide un menu si vor fi aratate commiturile de la cel mai vechi la cel mai nou
* Aici putem decide ce sa facem cu fiecare commit separat dupa ce acesta se va combina cu cel din branch scris.
* Daca stergem cu totul un commit de acolo, el nu va fi atins in vreun fel, si va fi sters cu totul dupa ce restul vor face merge.
* **pick** – este ceea ce face in mod default comanda git rebase Branch. Doar combina commiturile si atat.
* **r(reword)** - face tot aceeasi ca si pick, doar ca procesul de rebase va fi oprit si intai vom modifica mesajul noului commit creat, si apoi se va face combinarea.
* **edit(e)** – ne ofera posibilitatea de a da amend la commit, adica sa il modificam cum vrem.
* **s(squash) si f(fixup)** – fac ca contetul din commitul selectat sa fie unit cu cel de mai sus, adica modificarile salvate de el vor fi adaugate la cel de mai sus de el, caci doar cel curent deja le are pe ale celui precedent oarecum.Deci, e logic ca nu putem scrie squash la primul commit, asa cum deasupra lui nu e altul.Unica diferenta e ca squash e pentru a combina mesajele la commituri printr-un editor, dar fixup face ca **mesajul pentru commitul la care e scrisa comanda sa dispara si sa ramana mesajul la commitul de mai sus cu care se va contopi**.Sunt bune ca ne permit sa unim mai multe commituri in unul si asa e mai comod de lucrat cu unul doar. Asadar, commiturile vor disparea si se vor combina in unul.

!!!Daca vom folosi squash cand vom da rebase cu un commit din alt branch, intai se va combina cu commitul dat, apoi se va uni cu commitul de mai sus.De ex:

pick commit1

squash commit2

squash commit3

Unim cu commitB

Deci, intai commit1 se va uni cu commitB, apoi commit2 se va uni cu commit B si apoi noul commit se va uni cu commit1 rezultand, sa zicem commit12B, apoi commit3 se va uni cu commitB si apoi noul commit se va uni cu commit12B.

* Daca folosim squash, se va deschide un text editor pentru mesaj.
* Daca folosim squash sau fixup, vom vedea ca commitul din branch unde eram a disparut din git log

**Shift + i pentru a scrie**

**Esc pentru a termina**

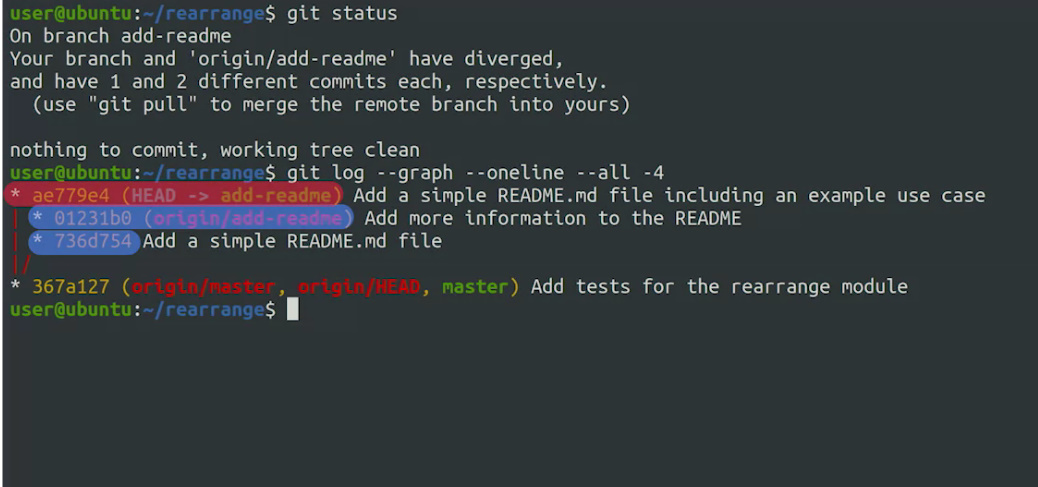
**:wq pentru a salva sau Shift Z**

**git rebase -i HEAD~3**

git rebase -i nu este folosit doar pe branches, dar poate fi folosit si pentru a face anumite modificari la commiturile din branch actual, fara a le combina cu vreo alta branch.

Pentru asta scriem git rebase -i HEAD~Numar

Head e branch actual, dar trebuie sa indicam cu ~ cate commituri din ea vrem sa editam.

* reword – doar modifica mesajul commiturlui si atat
* squash,fixup – doar unim commituri
* 

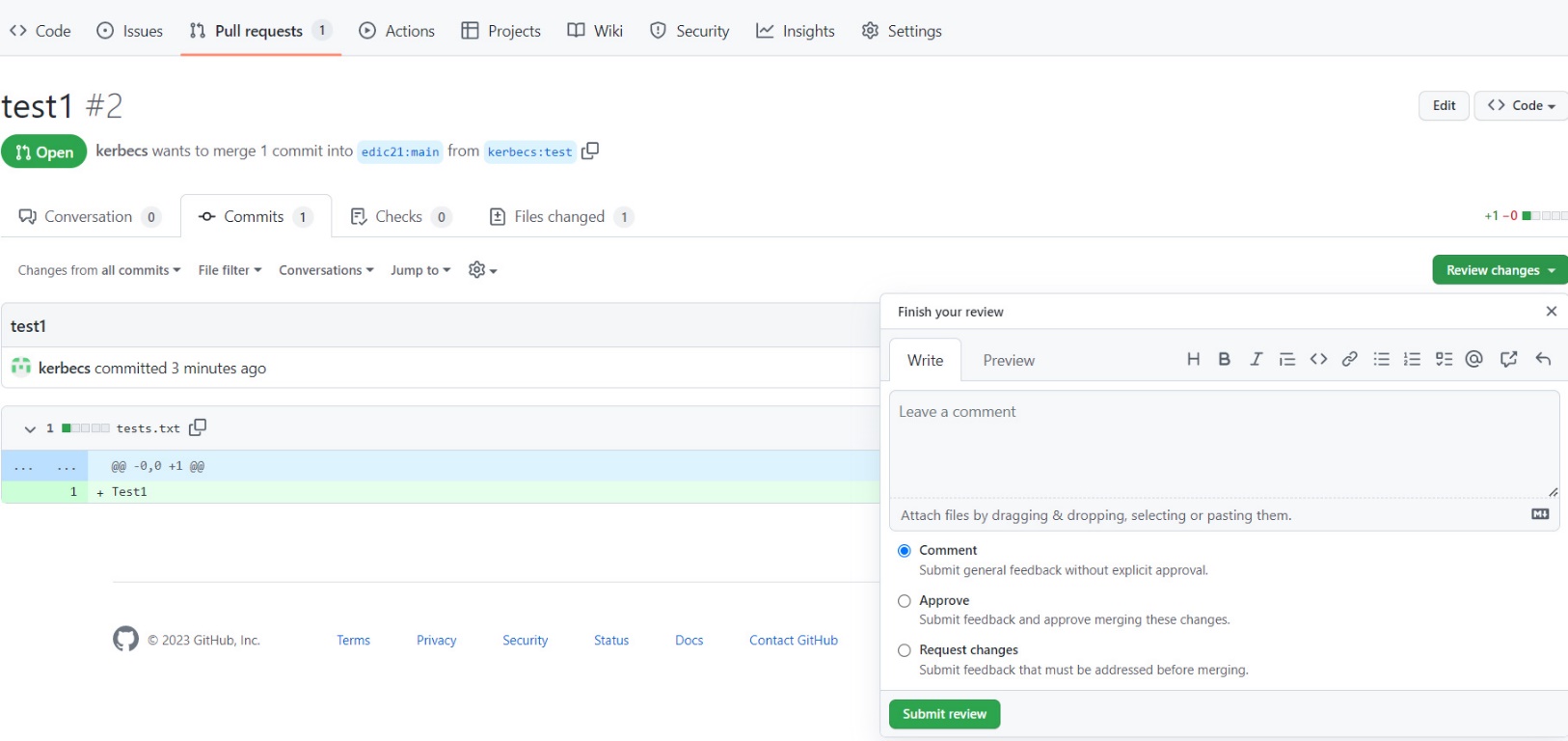
Vedem si in grafic ca acele 2 commituri nici nu mai au legatura cu cel final.

**git push -f**

* Nu facem nici-un merge, dar inlocuim commitul vechi cu cel nou, si asa fortam git push, chiar de sunt conflicte. Asa, commit trimis nu va face nici-un merge cu cel actual din remote, ci se va pune peste ele asa cum este, insa se va pierde legatura intre commituri.

**code reviews**

* Code review – trecem prin codul,documentatia,config altcuiva si verificam daca ele au sens, respecta regulile/patternurile necesare.
* Rolul lui este de a verifica modificarile pentru a fi siguri ca sunt calitative si corecte
* Acest tool ne permite sa ne uitam prin codul altcuiva, sa il verificam si sa comentam ce am depistat dupa verificare.
* Oricine poate face review la un project public de pe github, chiar si cei ce nu au nici-un acces de modificare la el. Asa ownerul are doar de castigat cand altii scriu ce nu e bine, sau daca totul e super.
* Cei care dau reviews pot scrie deci comentarii, noi le putem raspunde, sau face ce zic ei ca trebuie facut.
* Uneori, drept comentariu putem primi o intrebare, si deci raspundem pur si simplu la ea direct.
* **Review changes se face aici, cand ne uitam la ce a fost trimis de la un user in pull request:**

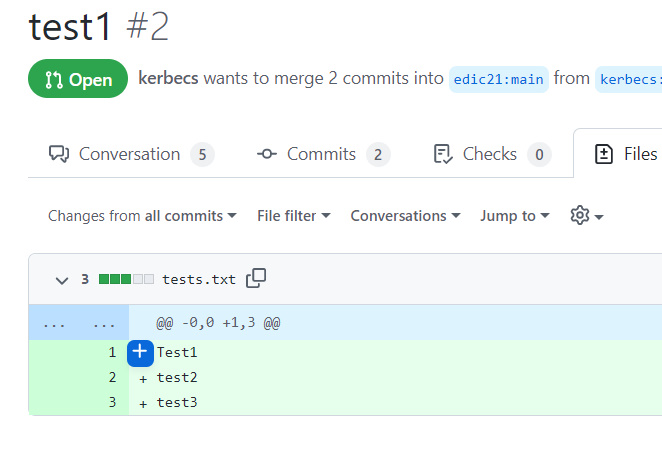


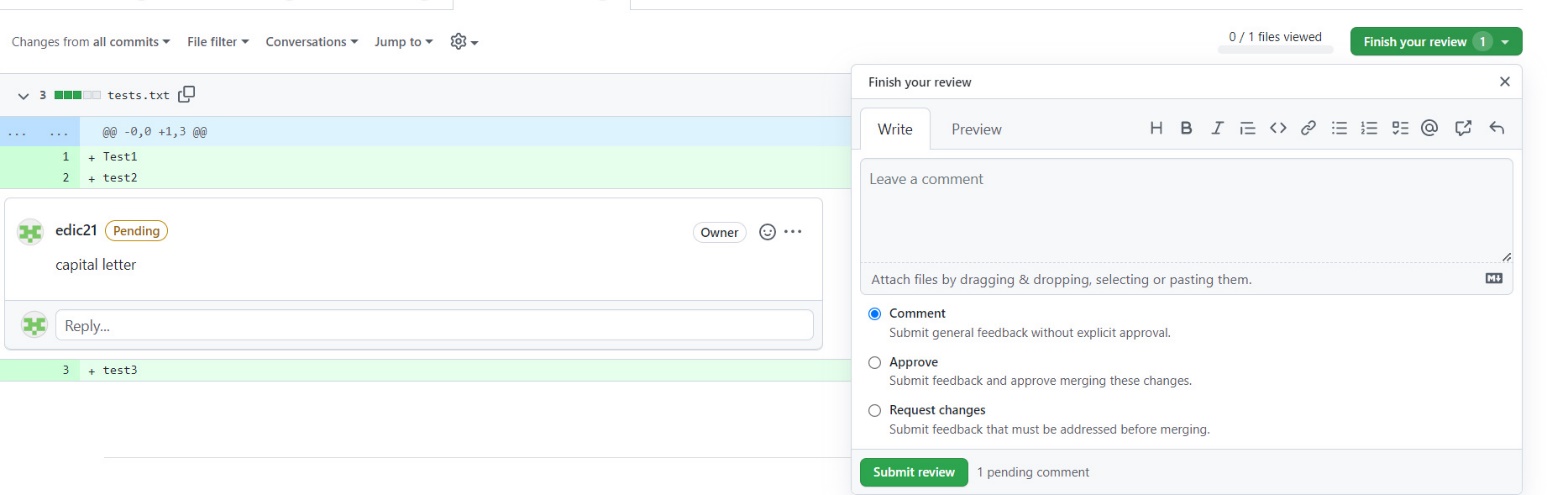
comment – facem un simplu comentariu, fara a aproba sau cere ceva

approve – aprobam schimbarile spre merge, dar nu se face merge inca. Va arata doar statusul de acceptat, adica ca am acceptat ce s-a facut

request changes – cerem niste modificari

* Putem da un review si unei singure linii sau mai multe, si anume aici cream reviews la code:



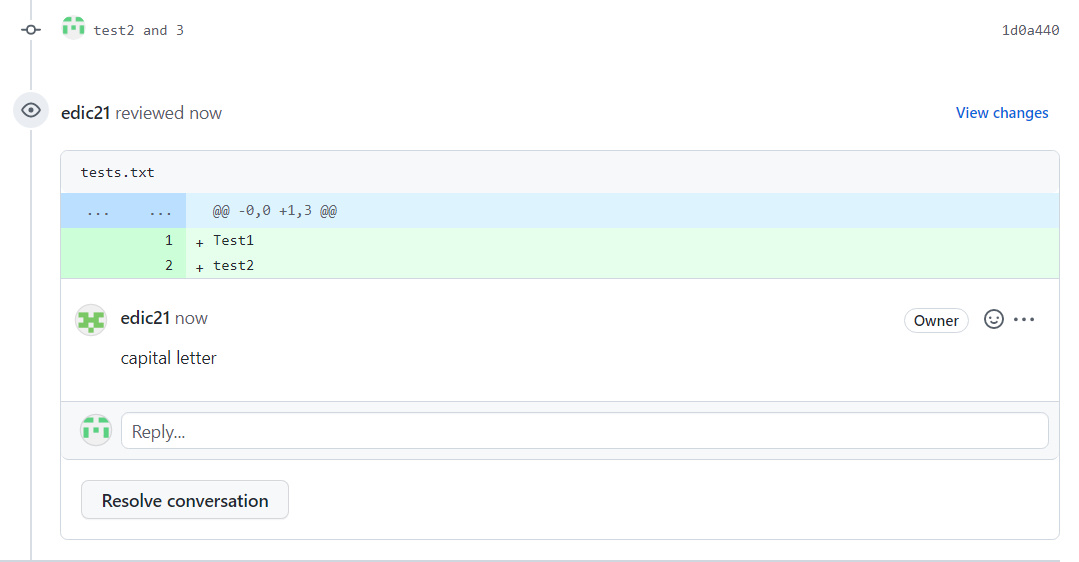




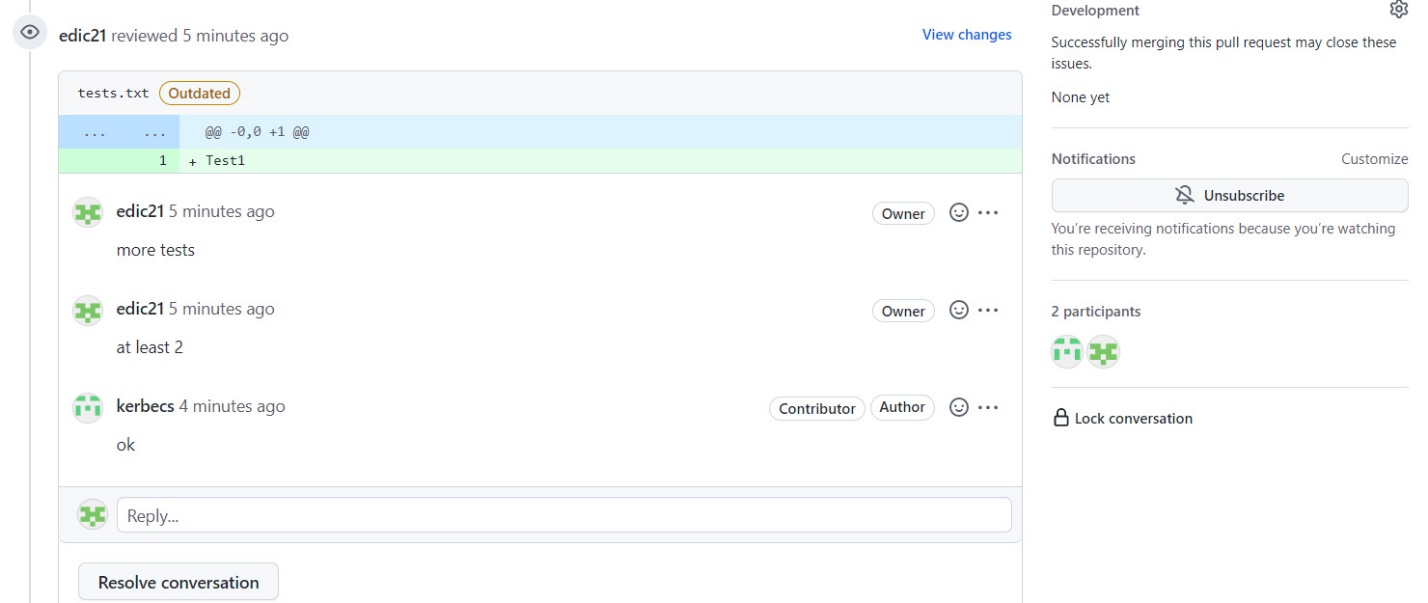
Apoi apasam Finish your review ca sa trimitem comentariile.



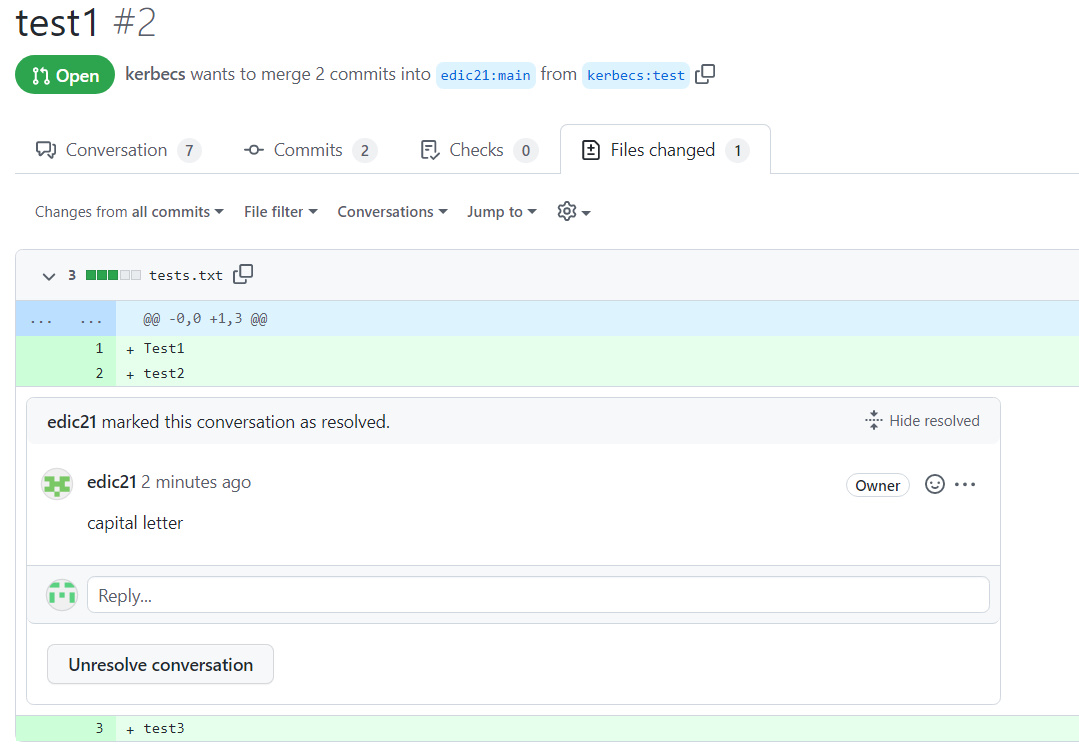
In menu va arata asa:



* Cand nu mai avem vreun review de facut si totul e bine, apasam pe Resolve conversation si incheiem conversatia inceputa la acel fisier/linie/etc..

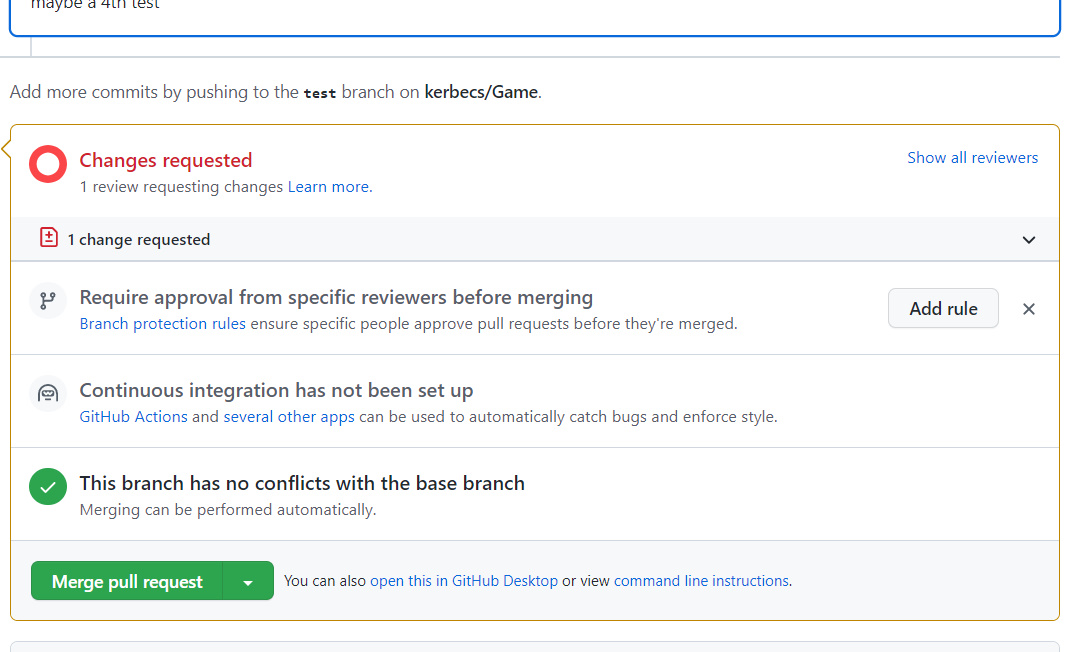


* O resolved conversation poate fi redeschisa oricand si de owner si de cotntribuitor:

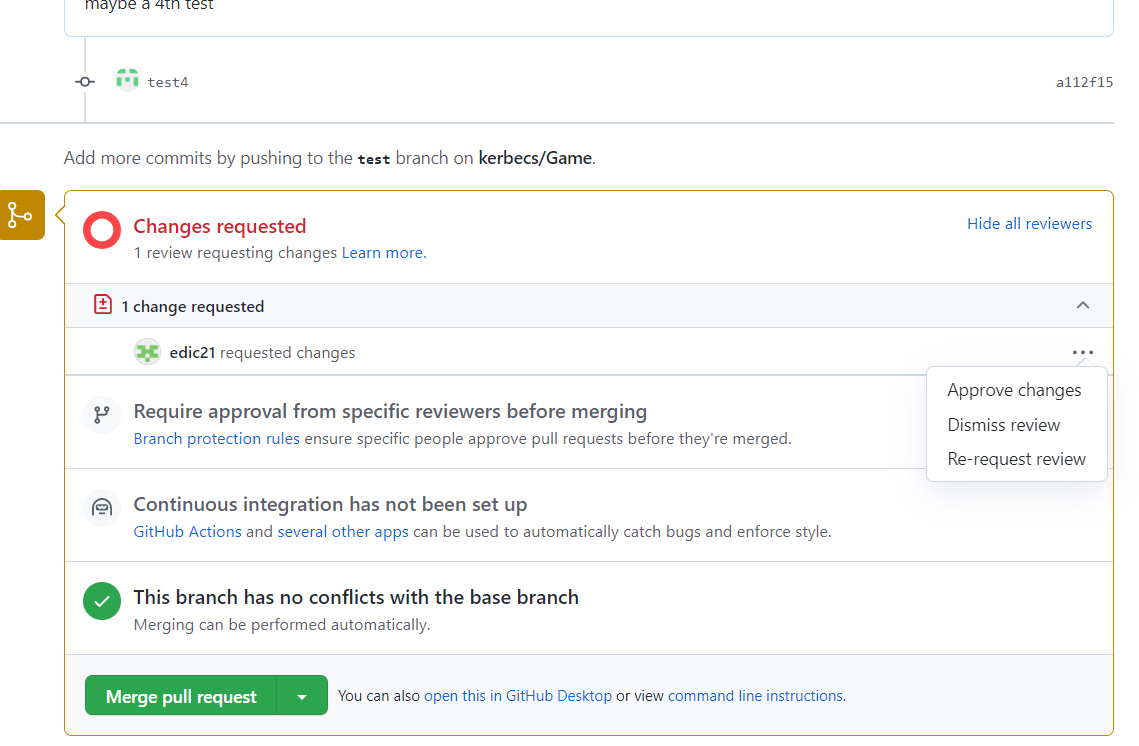
****



* Un request changes va arata asa:

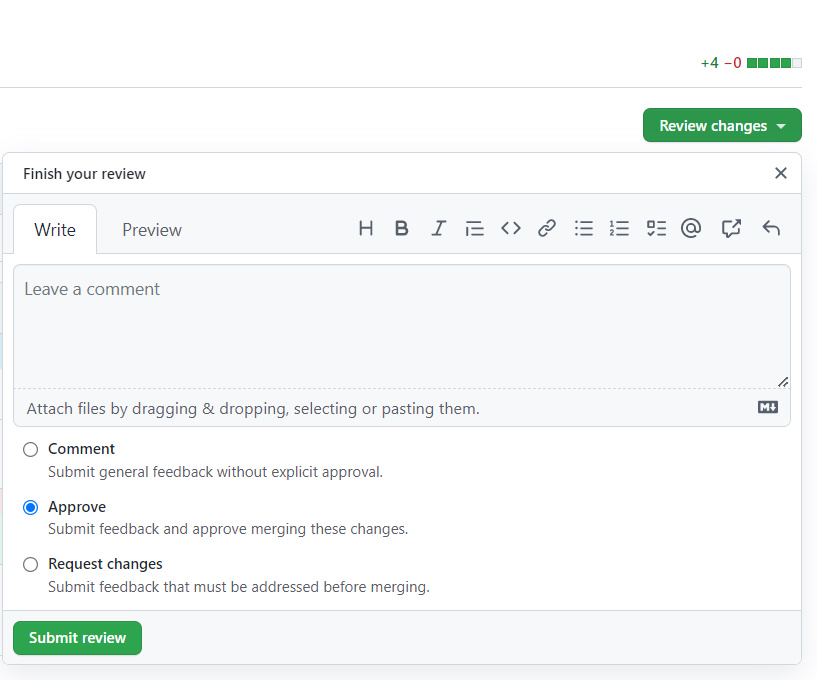
****

Putem sa la acceptam:



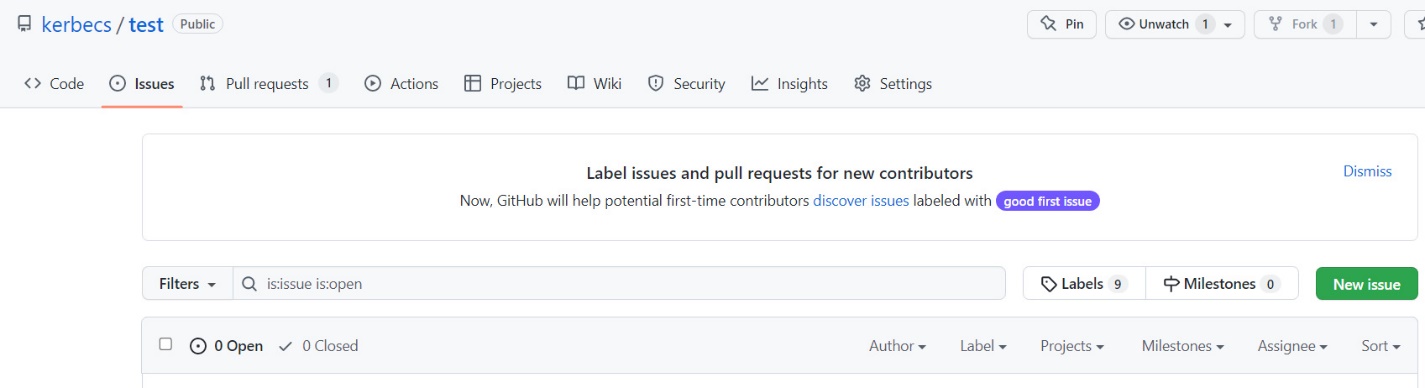


sau



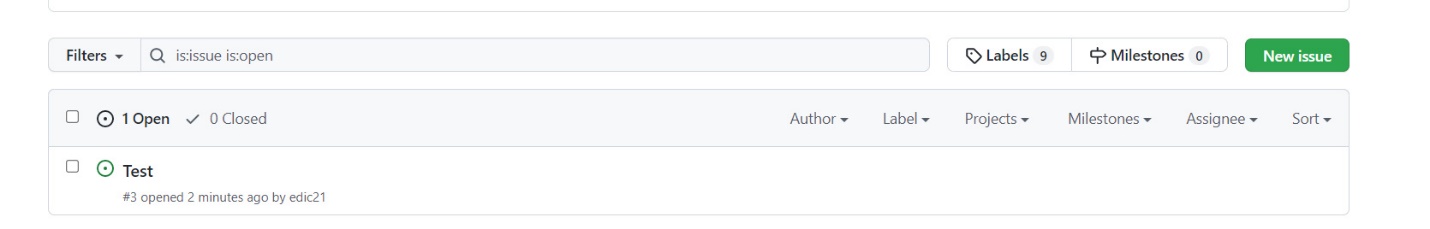
**Tracking issues**

* A decide cine si ce va face e foarte important cand lucreaza mai multi la un proiect.
* Totusi, cand in echipa sunt foarte multe persoane, impartirea repsonsabilitatilor devine dificila. Pentru a ne ocupa de asta, folosim un tool numit **issue tracker**
* **Issue tracker** – ne spune ce sarcini trebuie facute, statusul lor si cine lucreaza la ele. Putem adauga si comentarii la issue pentru a da detalii, si probleme aparute.
* Totodata, issues trackers nu sunt utile doar pentru coordonarea lucrului in echipa, dar sunt bune si prin faptul ca permit userilor sa raporteze buguri
* GitHub are un issue tracker incorporat



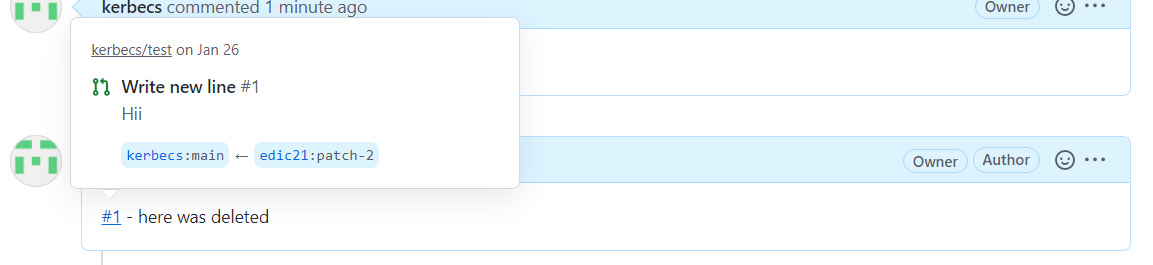


* Ca si pull request, issues au un id unic, fiind numarul lor de ordine, dar pull requests si issues mereu au id diferite, ca sa ne putem referi mai apoi la ele cumva.



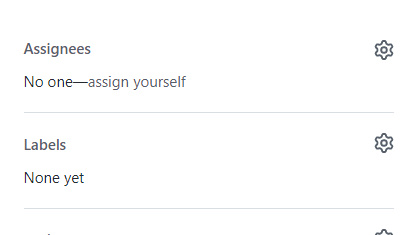


* Click pe new issue pentru a crea una. Contributorii pot comenta
* GitHub are automat referinte la ID al pull requests si issues, asa incat putem si noi sa ne referim la ele in comments, commits etc. prin hash tag number format
* De exemplu, daca intr-un commentariu vom scrie #1 si 1 e id la un pull request, automat se va crea o referinta la el.



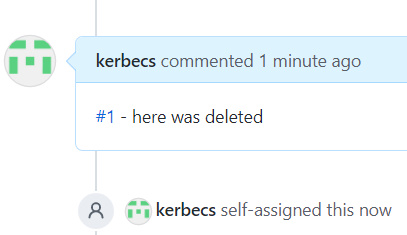


* Daca e vorba de un bug, cineva poate anunta ca el se va ocupa de el, prin apasarea butonului assign yourself:





Si asta va fi aratat asa:



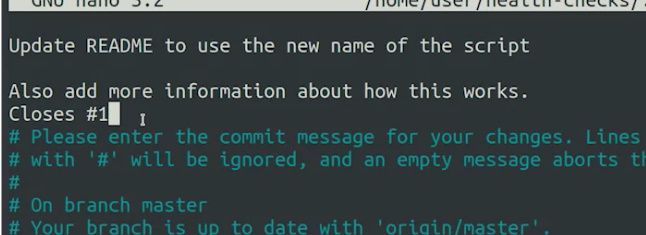
Asa restul vor sti ca noi lucram la aceasta issue(bug etc.)

* Daca rezolvam vreun issue printr-un pull request, putem inchide automat acea issue deschisa pe github odata ce codul e merged. E suficient in commentariul la pull request sau la commit trimis cu push sa folosim pe undeva:

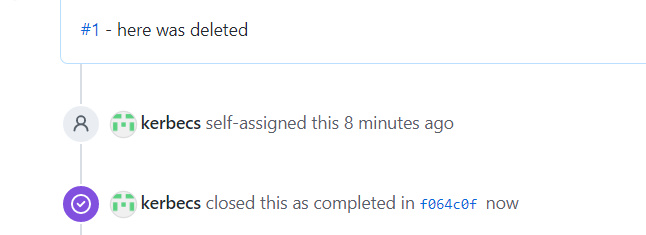
Closes #4 sau Closes : #4

Asa gitHub va vedea comanda asta in comment si va sti ca trebuie sa inchida automat acel issue. **Atentie! Aceste comenzi au efect doar asupra la branch principal din repository daca e vorba de pull request! Daca o scriem in alt branch in pull request, ea va fi pur si simplu ignorata.**

* Closes nu e unicul, mai sunt si fixes,fixed,closed etc.
* Odata ce github va vedea asta, el va inchide automat issue cu un mesaj ce va avea o referinta la acel commit. De ex:



Si asa va arata dupa git push:



**Continuous Integration**

* Adesea cand facem proiecte si scriem cod, uitam sa verificam daca totul functioneaza bine. Mai ales uitam sa facem teste
* Nici nu trebuie mereu sa ne amintim sa facem fiecare test. Putem scrie teste automate ce testeaza codul pentru noi si sa folosim un Continuous Integration System pentru a le rula automat.
* **Continuous integration system** – va crea si testa codul de fiecare data cand apare vreo modificare. Asta inseamna ca el va rula mereu cand e vreun nou commit in main branch sau cand ceva nou vine cu pull request de la altcineva
* Deci, daca avem continuous integration configurata in proiectul nostru, putem rula automat testele noastre folosind codul dintr-un pull request. In asa mod, putem verifica daca testul va trece dupa ce noile modificari vor fi merged in tree, indiferent ca testam pe ale noastre sau pe alte altora.Asa nu trebuie sa speram ca colaboratorii vor testa codul lor, dar putem sa il legam de sistemul nostru cu teste sa faca asta.
* **Continuous delivery(deployment) –** acesta este urmatorul pas care vine odata ce codul este testat si creat. GitHub nu ofera un tool pentru CD. El consta in faptul ca odata ce codul nou e testat, el este adaugat treptat in intregul proiect, nu tot deodata.Asa, posibilile errori sunt gasite si rezolvate mai devreme si mai usor. Pentru github avem **Travis**
* **Pipelines –** specifica pasii care trebuie urmati pentru a obtine rezultatul dorit.De ex: testeaza codul, compileaza-l, ruleaza, adauga pe server etc.
* **Artifacts –** nume folosit pentru a descrie orice fisiere care sunt generate ca parte din pipelines. Deobicei include compiled version of the code.