

Git & GitHub pentru developeri

Manual de curs · lecții și aplicații

Lectură estimată: 33 minute. Calculată din text; analiza codului, exercițiile și proiectele se adaugă separat.

De ce Git

Lectură estimată: 7 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Problema pe care o vei rezolva: Conflict de Git cu recuperare

Repository de laborator cu README. Branchul main schimbă instrucțiunea de instalare, branchul feature schimbă aceeași linie și adaugă un exemplu. Nu folosi un repository de producție.

Rezultatul urmărit

Creează conflictul, citește ambele intenții și păstrează instrucțiunea corectă plus exemplul. Înregistrează comiturile, apoi exersează recuperarea unui commit pe un branch separat folosind reflog.

Cum recunoști o soluție corectă

Conflictul este rezolvat fără marcaje rămase și fără pierderea exemplului. Commitul recuperat este referit de un branch. Istoricul partajat nu este rescris pentru a simplifica exercițiul.

Parcurge lecțiile în ordine și păstrează exercițiile în registrul descărcabil. Ultima lecție include atelierul de 40 de minute în care combini conceptele și testezi o variație a situației inițiale. Exemplele comerciale sunt fictive; nu reprezintă rezultate ale unor clienți.

Înainte să înveți comenzi, merită să înțelegi ce problemă rezolvă Git. Altfel o să memorezi git add și git commit fără să știi de ce le scrii, iar în prima situație neprevăzută o să rămâi blocat.

Aproape toți oamenii care ajung la Git au trecut întâi prin varianta manuală. Ai un folder cu proiectul, faci o modificare mai mare și, ca să fii sigur că nu strici ce funcționează, copiezi folderul. Așa apar site, site-copie, site-final, site-final-2 și, inevitabil, site-final-ASTA-E-BUNA.

Funcționează până într-o zi. Ziua aceea vine când trebuie să răspunzi la o întrebare simplă: ce s-a schimbat între versiuni și de ce?

Ce este de fapt Git

Git este un sistem de control al versiunilor. În loc să păstrezi copii întregi ale folderului, Git păstrează istoricul modificărilor: cine a schimbat, ce a schimbat, când și — dacă scrii mesaje decente — de ce.

Concret, Git îți dă patru lucruri pe care copierea de foldere nu ți le dă niciodată:

- Istoric complet. Poți vedea orice stare prin care a trecut proiectul și te poți întoarce la ea.
- Context. Fiecare modificare are autor, dată și mesaj. Peste șase luni, mesajul acela e singurul lucru care îți explică de ce codul arată ciudat.
- Lucru în paralel. Poți începe o funcționalitate nouă fără să atingi versiunea care merge în producție.
- Colaborare fără să vă călcați pe picioare. Doi oameni pot lucra în același fișier, iar Git știe să combine rezultatul.

Git funcționează local, pe calculatorul tău. Nu ai nevoie de internet și nu ai nevoie de GitHub ca să folosești Git. Asta e o confuzie extrem de des întâlnită la început, așa că hai să o lămurim acum.

Git nu este GitHub

Git este programul care ține istoricul. GitHub este un serviciu online unde poți găzdui repository-uri Git, ca să le ai într-un loc sigur și ca să poată ajunge și alți oameni la ele.

Git e motorul. GitHub e parcare publică unde îți ții mașina și unde o pot vedea și alții. Poți avea motor fără parcare. Parcare, fără motor, nu te duce nicăieri.

Alternative la GitHub există: GitLab, Bitbucket, Gitea, Codeberg. Toate vorbesc același Git. Ce înveți aici funcționează la fel peste tot; se schimbă doar interfața web și denumirile unor butoane.

Cele trei zone pe care trebuie să le ai în cap

Dacă reții un singur lucru din lecția asta, reține diagrama următoare. Aproape toate confuziile de început vin din faptul că oamenii nu știu în care zonă se află modificarea lor.

```
Working directory → Staging area → Repository
(fișierele tale)    (git add)         (git commit)
```

- Working directory — folderul în care editezi. Aici există modificările tale nesalvate în istoric.
- Staging area (sau „index”) — o zonă intermediară în care pui modificările pe care vrei să le incluzi în următorul commit. Aici e diferența față de alte sisteme: poți alege o parte din ce ai modificat.
- Repository — istoricul propriu-zis, folderul ascuns .git. Ce ajunge aici e salvat permanent.

Staging area pare o complicație inutilă la început. Nu e. Ea îți permite să faci ordine: dacă ai reparat un bug și, între timp, ai schimbat și niște spații, poți commite separat bug-ul, ca istoricul să rămână citibil.

Instalare și prima configurare

Pe Windows instalezi Git for Windows, care îți aduce și Git Bash — un terminal în care funcționează comenzile din curs. Pe macOS vine cu Xcode Command Line Tools (xcode-select --install), iar pe Linux îl iei din managerul de pachete (sudo apt install git).

Verifici instalarea:

```
git --version
# git version 2.47.0
```

Apoi spui cine ești. Numele și emailul ajung în fiecare commit pe care îl faci, deci pune date reale:

```
git config --global user.name "Numele Tău"
git config --global user.email "adresa@exemplu.ro"
```

Mai sunt trei setări care îți scutesc mult timp mai târziu:

```
# branch-ul principal se va numi main, nu master
git config --global init.defaultBranch main

# editorul folosit pentru mesaje de commit (exemplu: VS Code)
git config --global core.editor "code --wait"

# pe Windows: normalizează sfârșiturile de linie
git config --global core.autocrlf true
```

Ultima merită explicată. Windows termină liniile cu CRLF, Linux și macOS cu LF. Fără setarea asta, o să vezi diff-uri în care „s-a schimbat tot fișierul”, deși nimeni nu a atins nimic. Este una dintre cele mai frecvente frustrări ale începătorilor pe Windows.

Poți verifica oricând ce ai configurat:

```
git config --list --show-origin
```

Ce înseamnă un commit bun

Un commit este o fotografie a proiectului la un moment dat, însoțită de un mesaj. Calitatea istoricului tău este dată aproape exclusiv de calitatea acestor mesaje.

Regula practică: mesajul răspunde la întrebarea „ce face acest commit?”, la timpul prezent, pe prima linie, în maximum 50-72 de caractere.

```
# slab  
update  
fix  
modificari  
asdf
```

```
# bun  
Adaugă validare pentru formularul de contact  
Repară calculul TVA la produsele cu reducere  
Elimină scriptul de analytics neutilizat
```

Dacă ai nevoie de context suplimentar (de ce ai ales soluția asta, ce alternative ai respins, ce bug reproduce), lași o linie goală și scrii un paragraf după titlu. Nu e birocrație: peste un an, tu vei fi cel care citește.

Cât de des faci commit

Regula bună: un commit = o schimbare logică completă.

Prea rar (un commit pe zi, cu 40 de fișiere) înseamnă că nu mai poți izola nimic și nu poți da înapoi o singură decizie. Prea des (un commit per literă scrisă) înseamnă zgomot în istoric.

Întrebarea de control: dacă mâine trebuie să anulez exact această schimbare, pot să o fac fără să distrug altceva? Dacă răspunsul e da, e un commit bun.

De ce contează asta chiar dacă lucrezi singur

Mulți începători cred că Git e util doar în echipă. Realitatea e că cele mai mari beneficii apar și când lucrezi singur:

- poți experimenta fără frică, pentru că întotdeauna te poți întoarce;
- poți afla când s-a stricat ceva care funcționa (git log, git bisect);
- ai o copie de siguranță reală în momentul în care trimiți pe GitHub;
- îți construiești un portofoliu public, dacă repository-urile sunt deschise.

Exercițiu

- Instalează Git și rulează git --version.
- Configurează user.name, user.email și init.defaultBranch.
- Rulează git config --list și verifică valorile.
- Scrie pe hârtie trei mesaje de commit pentru ultimele trei modificări pe care le-ai făcut într-un proiect. Verifică dacă respectă regula de mai sus.

Ce urmează

În lecția următoare creăm primul repository și trecem efectiv prin ciclul status → add → commit → log, adică fluxul pe care îl vei repeta de mii de ori de acum încolo.

Aplicație practică

Sarcina

Creează două versiuni ale unui fișier și inspectează diferența.

Rezolvare orientativă și verificare

git diff arată modificarea înainte de commit; git log păstrează istoricul după commit.

Greșeală de evitat

Git nu este backup pentru fișiere ignorate și secrete locale.

Documentație și surse

- Pro Git — Git

Primul repository

Lectură estimată: 4 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

În lecția asta faci ciclul complet: creezi un repository, faci primele commit-uri, citești istoricul și trimiți totul pe GitHub. Deschide un terminal și lucrează efectiv, nu doar citi — Git se învață cu mâinile.

Creezi repository-ul

```
mkdir proiect-test
cd proiect-test
git init
```

Git îți răspunde ceva de genul Initialized empty Git repository in .../proiect-test/.git/. Acel folder .git este tot repository-ul: istoric, configurări, referințe. Dacă îl ștergi, rămâi cu fișierele, dar pierzi istoricul.

Creezi un fișier și vezi ce spune Git:

```
echo "# Proiect test" > README.md
git status

On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  README.md
```

Untracked înseamnă că Git vede fișierul, dar nu îl urmărește încă. Ia-ți obiceiul să rulezi git status des; e comanda care îți spune mereu unde te afli.

add și commit

```
git add README.md
git status
```

Acum fișierul apare sub Changes to be committed. E în staging area. Îl salvezi în istoric:

```
git commit -m "Adaugă README cu descrierea proiectului"

[main (root-commit) 8f3a1c2] Adaugă README cu descrierea proiectului
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

8f3a1c2 este începutul hash-ului commit-ului — identificatorul lui unic. Îl vei folosi când vrei să te referi la o stare anume.

Al doilea commit și diferențele

Modifică fișierul și adaugă altul:

```
echo "Proiect de test pentru cursul de Git." >> README.md
echo "console.log('salut');" > index.js
git status
```

Acum ai două situații diferite: README.md e modified (Git îl urmărește deja), iar index.js e untracked. Înainte să dai add, vezi exact ce s-a schimbat:

```
git diff
```

```
diff --git a/README.md b/README.md
index 9f2a0e1..3c1b7d4 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
 # Proiect test
+Proiect de test pentru cursul de Git.
```

Liniiile cu + sunt adăugate, cele cu - sunt șterse. git diff arată ce e modificat dar nestaged. Dacă vrei să vezi ce e deja în staging, folosești git diff --staged. Confuzia dintre cele două e clasică.

Adaugi tot și commiti:

```
git add .
git commit -m "Adaugă scriptul de intrare și detaliază README"
```

Atenție la git add .: adaugă tot ce e în folderul curent, inclusiv lucruri pe care nu vrei să le urci. De aceea avem .gitignore.

.gitignore — ce nu urci niciodată

Unele fișiere nu au ce căuta în istoric: dependențe instalate, fișiere de configurare cu parole, fișiere generate, gunoi de la sistemul de operare.

Creezi un fișier .gitignore în rădăcina proiectului:

```
/node_modules
/vendor
/.env
/storage/*.key
/public/build
.DS_Store
Thumbs.db
*.log
```

Regula de aur: fișierele cu secrete (.env, chei API, parole) nu ajung niciodată în Git. Odată ce ai comis un secret și ai dat push, consideră-l compromis — istoricul rămâne accesibil chiar dacă ștergi fișierul într-un commit ulterior. Singura reacție corectă e să schimbi cheia.

Important: .gitignore funcționează doar pentru fișiere pe care Git nu le urmărește deja. Dacă ai comis din greșeală .env, trebuie mai întâi să-l scoți din urmărire:

```
git rm --cached .env
git commit -m "Elimină .env din versionare"
```

Citirea istoricului

```
git log
```

E stufos. Varianta pe care o vei folosi zilnic:

```
git log --oneline --graph --decorate -10

* 4b7e9a1 (HEAD -> main) Adaugă scriptul de intrare și detaliază README
* 8f3a1c2 Adaugă README cu descrierea proiectului
```

Merită să îți faci un alias, ca să nu scrii toate opțiunile de fiecare dată:

```
git config --global alias.lg "log --oneline --graph --decorate -20"
# de acum: git lg
```

Ca să vezi ce s-a schimbat exact într-un commit:

```
git show 8f3a1c2
```

Trimiterea pe GitHub

Creezi pe GitHub un repository gol (fără README, fără .gitignore — le ai deja local, iar altfel vei avea de rezolvat un conflict inutil din prima). Apoi legi repository-ul local de cel remote:

```
git remote add origin https://github.com/utilizator/proiect-test.git
git branch -M main
git push -u origin main
```

- origin e doar un nume, convenția pentru „remote-ul principal”.
- -u leagă branch-ul local de cel remote, ca pe viitor să poți scrie doar git push.

Verifici ce remote-uri ai:

```
git remote -v
```

Autentificare: parola nu mai merge

GitHub nu mai acceptă parola contului la push. Ai două variante:

- Personal Access Token — îl generezi din Settings → Developer settings → Personal access tokens și îl folosești în loc de parolă. Git Credential Manager îl reține.
- Cheie SSH — varianta comodă pe termen lung:

```
ssh-keygen -t ed25519 -C "adresa@exemplu.ro"
cat ~/.ssh/id_ed25519.pub
# copiezi rezultatul în GitHub → Settings → SSH and GPG keys
ssh -T git@github.com
```

Cu SSH, URL-ul remote arată așa: git@github.com:utilizator/proiect-test.git.

Clonarea unui proiect existent

Când pornești de la un proiect care există deja, nu faci git init, ci:

```
git clone https://github.com/utilizator/proiect.git
cd proiect
```

Clonarea aduce tot istoricul și configurează automat origin.

Exercițiu

- Creează un repository local nou și fă trei commit-uri cu mesaje corecte.
- Adaugă un .gitignore care ignoră node_modules și .env. Creează cele două și verifică cu git status că nu apar.
- Publică repository-ul pe GitHub și confirmă că vezi cele trei commit-uri în interfața web.
- Rulează git log --oneline --graph și compară cu ce vezi pe GitHub.

Aplicație practică

Sarcina

Inițializează un repository cu README și .gitignore.

Rezolvare orientativă și verificare

Verifică git status înainte de primul commit. Fișierul .env trebuie ignorat, iar .env.example să conțină doar valori demonstrative.

Greșeală de evitat

Adăugarea ulterioară în .gitignore nu șterge un secret din istoric.

Documentație și surse

- Pro Git — Git

Branch-uri

Lectură estimată: 4 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Branch-urile sunt motivul pentru care Git a câștigat în fața sistemelor mai vechi. În alte sisteme, o ramură era o operațiune grea, care dura minute și pe care o evitai. În Git, un branch este atât de ieftin încât poți face unul pentru o idee pe care o testezi zece minute.

Ce este, de fapt, un branch

Un branch în Git nu este o copie a proiectului. Este doar un indicator care arată către un commit. Când faci un commit nou, indicatorul se mută automat pe el.

```
A---B---C    (main)
      \
      D---E    (functionalitate-noua)
```

Mai există un indicator special: HEAD, care arată către branch-ul pe care te afli acum. De aceea în git log vezi HEAD -> main.

Fiindcă un branch e doar un pointer, crearea lui e instantanee, indiferent cât de mare e proiectul.

Comenzile de bază

```
# vezi branch-urile locale
git branch

# creezi un branch nou
git branch functionalitate-noua

# te muți pe el
git switch functionalitate-noua

# creezi și te muți dintr-o mișcare
git switch -c functionalitate-noua
```

Probabil ai văzut prin tutoriale git checkout -b. Face același lucru. git checkout este comanda veche, care făcea prea multe lucruri diferite (schimba branch-uri, dar și restaura fișiere), așa că Git a împărțit-o în două comenzi mai clare:

- git switch — pentru schimbat branch-uri;
- git restore — pentru restaurat fișiere.

Ambele funcționează; în curs folosim variantele noi, mai greu de confundat.

Cum numești branch-urile

Folosește un prefix și o descriere scurtă cu cratime:

```
feature/export-facturi-pdf
fix/calcul-tva-reducere
chore/actualizare-dependinte
docs/ghid-instalare
```

Numele contează în echipă: când sunt 30 de branch-uri deschise, „test2” și „andrei-final” nu spun nimic nimănui.

Fluxul normal de lucru

Regula practică într-un proiect sănătos: nu lucrezi niciodată direct pe main. main rămâne versiunea care funcționează.

```
# pleci mereu de la un main actualizat
git switch main
git pull

# deschizi un branch pentru sarcina ta
git switch -c feature/export-facturi-pdf

# lucrezi normal: editezi, add, commit (de mai multe ori)
git add .
git commit -m "Adaugă generarea PDF pentru facturi"

# publici branch-ul pe remote
git push -u origin feature/export-facturi-pdf
```

Ce se întâmplă cu fișierele când schimbi branch-ul

Când faci git switch, Git rescrie fișierele din folderul tău ca să reflecte starea branch-ului respectiv. Asta surprinde mulți începători: fișierele „dispar” sau „se schimbă singure”. Nu e magie, e exact ce ai cerut.

Dacă ai modificări necommise care intră în conflict cu branch-ul destinație, Git refuză să te mute și îți spune de ce. Ai trei opțiuni:

- commiti modificările;
- le arunci, dacă nu îți trebuie: git restore .;
- le pui deoparte temporar cu git stash.

git stash — pauza salvatoare

Scenariu tipic: ești în mijlocul unei funcționalități și cineva îți cere urgent o reparație în producție.

```
# pui deoparte tot ce ai nesalvat
git stash push -m "export pdf, in lucru"

# rezolvi urgența
git switch main
git switch -c fix/eroare-500-checkout
# ... repari, commiti, push ...

# te întorci și îți iei lucrul înapoi
git switch feature/export-facturi-pdf
git stash pop
```

Comenzi utile: git stash list (ce ai pus deoparte), git stash show -p (ce conține), git stash drop (arunci un stash).

Atenție: implicit, git stash nu include fișierele untracked. Pentru ele folosești git stash -u. E o sursă frecventă de „mi-am pierdut fișierul nou”.

Branch-uri locale vs. remote

Branch-urile de pe server sunt separate de cele locale. Când rulezi git fetch, Git aduce informația despre ele fără să îți schimbe fișierele:

```
git fetch
git branch -a      # toate branch-urile, inclusiv cele remote
git switch feature/ceva-facut-de-coleg # Git creează automat branch local legat de origin
```

Diferența dintre fetch și pull: fetch doar aduce informația, pull aduce și integrează (fetch + merge). Când vrei să vezi ce s-a schimbat înainte să integrezi, folosești fetch.

Ce ai în branch și nu ai în main

Înainte să deschizi un pull request sau să faci merge, merită să vezi exact ce aduci. Trei comenzi îți răspund la asta:

```
# commit-urile din branch care nu sunt în main
git log main..feature/export-facturi --oneline

# fișierele modificate față de main
git diff main --stat

# ce e în main și îți lipsește ție
git log feature/export-facturi..main --oneline
```

Notăția cu două puncte se citește „de la ... până la”. main..branch înseamnă „ce are branch-ul în plus față de main”. Inversând ordinea, afli dacă ai rămas în urmă și trebuie să te sincronizezi.

O verificare rapidă a branch-urilor deja integrate, utilă la curățenie:

```
git branch --merged main      # se pot șterge în siguranță
git branch --no-merged main   # mai au muncă neintegrată
```

Un scenariu real, cap-coadă

Ești la lucru și primești două sarcini: o funcționalitate nouă și, la mijlocul ei, un bug urgent în producție. Așa arată traseul complet:

```
git switch main && git pull
git switch -c feature/filtru-comenzi
# ... lucrezi, două commit-uri ...

# apare urgența
git stash push -u -m "filtru comenzi, la jumătate"
git switch main && git pull
git switch -c fix/total-comanda-gresit
# ... repari, comiți, push, pull request ...

# te întorci la ce făceai
git switch feature/filtru-comenzi
git stash pop

# aduci în branch reparația care între timp a intrat în main
git fetch origin
git merge origin/main
```

Observă ultimul pas: după ce reparația a ajuns în main, o aduci și în branch-ul tău. Altfel lucrezi peste un cod pe care știi deja că îl vei suprascrie la merge.

Curățenie

După ce un branch a fost integrat, îl ștergi ca să nu se adune:

```
# local
git branch -d feature/export-facturi-pdf

# remote
git push origin --delete feature/export-facturi-pdf

# ștergi referințele locale către branch-uri care nu mai există pe server
git fetch --prune
```

-d refuză să șteargă un branch neintegrat — e o plasă de siguranță. -D forțează ștergerea; folosește-l doar când ești sigur.

Exercițiu

- În repository-ul de test, creează branch-ul feature/pagina-contact și adaugă acolo un fișier nou, cu două commit-uri.
- Întoarce-te pe main și verifică: fișierul nu trebuie să existe.
- Modifică ceva pe main fără să commiti, apoi încearcă git switch. Rezolvă situația folosind git stash.
- Publică branch-ul pe GitHub și verifică-l în interfața web.

Aplicație practică

Sarcina

Creează un branch pentru schimbarea unei pagini.

Rezolvare orientativă și verificare

Fă commit pe branch și revino pe ramura principală. Verifică că modificarea nu apare până la integrare.

Greșeală de evitat

Lucrul necomis poate fi transportat între branch-uri în anumite situații.

Documentație și surse

- Pro Git — Git

Merge și conflicte

Lectură estimată: 4 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Ai lucrat pe un branch, ai terminat. Acum trebuie să aduci munca înapoi în main. Operațiunea se numește merge, iar frica de ea vine dintr-un singur cuvânt: conflict.

Vestea bună: un conflict nu este o eroare și nu ai stricat nimic. Este doar Git care spune, corect: „aici doi oameni au schimbat aceeași linie, nu am de unde să știu care versiune e bună — decide tu”.

Merge simplu

```
git switch main
git pull
git merge feature/pagina-contact
```

Dacă în main nu s-a întâmplat nimic între timp, Git face un fast-forward: mută pur și simplu indicatorul main înainte, pentru că nu e nimic de combinat.

Înainte: A---B---C (main)
 \
 D---E (feature)

După: A---B---C---D---E (main, feature)

Dacă între timp au apărut commit-uri și pe main, Git creează un merge commit, un commit cu doi părinți:

A---B---C---F---M (main)
 \
 D---E (feature)

Anatomia unui conflict

Conflictul apare doar când aceeași porțiune din același fișier a fost modificată în ambele părți. Dacă tu ai lucrat în header.php și colegul în footer.php, Git combină fără să te întrebe nimic.

Când apare, Git îți spune clar:

```
Auto-merging resources/views/contact.blade.php
CONFLICT (content): Merge conflict in resources/views/contact.blade.php
Automatic merge failed; fix conflicts and then commit the result.
```

Deschizi fișierul și găsești marcajele:

```
<<<<<< HEAD
<h1>Contact</h1>
<p>Scrie-ne la contact@exemplu.ro</p>
=====
<h1>Ia legătura cu noi</h1>
<p>Telefon: 0700 000 000</p>
>>>>>> feature/pagina-contact
```

- între <<<<<< HEAD și ===== e versiunea din branch-ul în care faci merge (de regulă main);
- între ===== și >>>>>> e versiunea care vine.

Cum rezolvi

Rezolvarea înseamnă să editezi fișierul până arată cum trebuie și să ștergi complet marcasele. Rezultatul poate fi una dintre versiuni, sau o combinație:

```
<h1>Ia legătura cu noi</h1>
<p>Scrie-ne la contact@exemplu.ro</p>
<p>Telefon: 0700 000 000</p>
```

Apoi marchezi fișierul ca rezolvat și finalizezi:

```
git add resources/views/contact.blade.php
git status          # vezi dacă mai sunt fișiere în conflict
git commit          # mesajul de merge e precompletat
```

Dacă te-ai încurcat și vrei să o iei de la capăt:

```
git merge --abort
```

Comanda te întoarce exact în starea de dinaintea merge-ului. E plasa ta de siguranță; folosește-o fără ezitare.

Greșeala numărul unu

Cea mai frecventă greșeală la conflicte este să uiți marcasele în fișier. Codul ajunge în producție cu <<<<<< HEAD în el și pagina se rupe.

Verificare rapidă înainte de commit:

```
git diff --check
grep -rn "<<<<<<" .
```

Unelte care te ajută

Nu trebuie să rezolvi conflictele în text brut. VS Code afișează butoane „Accept Current / Accept Incoming / Accept Both” direct în fișier. Există și:

```
git mergetool          # deschide unealta configurată
git checkout --ours fisier.txt  # păstrează versiunea din branch-ul curent
git checkout --theirs fisier.txt # păstrează versiunea care vine
```

Atenție la --ours / --theirs: aruncă complet cealaltă variantă. Sunt utile pentru fișiere generate (de exemplu un build), nu pentru cod scris de oameni.

Cum eviți conflictele mari

Conflictele nu se rezolvă cu comenzi mai deștepțe, ci cu obiceiuri mai bune:

- Branch-uri scurte. Un branch care trăiește trei săptămâni garantează durere. Unul de două zile, aproape niciodată.
- Sincronizează des. Adu main în branch-ul tău periodic, nu doar la final.
- Împarte munca pe fișiere diferite, când e posibil.
- Nu reformata fișiere întregi în commit-uri care conțin și logică. Un „prettier pe tot proiectul” transformă orice merge într-un coșmar.

```
# în branch-ul tău, periodic:
git fetch origin
git merge origin/main
```

Conflicte în fișiere care nu sunt cod

Există două categorii de fișiere care produc conflicte urâte și pe care merită să le tratezi separat.

Fișierele de dependențe (package-lock.json, composer.lock) nu se rezolvă editând manual. Iei versiunea de pe main și regenerezi:

```
git checkout --theirs package-lock.json
npm install
git add package-lock.json
```

Migrările de bază de date nu produc conflict de text, dar produc conflict de realitate: doi colegi creează migrări cu aceeași tabelă sau modifică aceeași coloană. Git le va combina fără să se plângă, iar problema apare abia la rulare. Verifică întotdeauna ce migrări noi vin odată cu merge-ul:

```
git diff main --stat -- database/migrations
```

Ce faci când merge-ul aduce ceva stricat

Se întâmplă: ai făcut merge, ai împins, iar pe main ceva nu mai merge. Dacă merge-ul e deja public, nu rescrii istoricul, ci îl anulezi ca pe orice commit — cu o mențiune importantă, pentru că un merge commit are doi părinți și Git trebuie să știe la care dintre ei să se întoarcă:

```
git revert -m 1 <hash-ul-merge-commit-ului>
```

-m 1 înseamnă „păstrează linia principală”, adică branch-ul în care ai integrat (de obicei main). Reține că, după un astfel de revert, dacă vrei să reintegrezi mai târziu același branch, va trebui să anulezi întâi revert-ul — altfel Git consideră că modificările au fost deja aduse.

Exercițiu (provoacă un conflict intenționat)

- Pe main, modifică prima linie din README și commit.
- Creează feature/titlu pornind de la commit-ul anterior, modifică aceeași linie diferit și commit.
- Fă merge în main și rezolvă conflictul păstrând o combinație a celor două variante.
- Repetă, dar de data asta folosește git merge --abort ca să te întorci.

Aplicație practică

Sarcina

Produce un conflict controlat în același rând din README.

Rezolvare orientativă și verificare

Citește ambele variante, rezolvă sensul, elimină marker-ele și rulează verificarea proiectului.

Greșeală de evitat

Acceptarea automată a unei părți poate pierde modificări necesare.

Documentație și surse

- Pro Git — Git

Rebase pe scurt

Lectură estimată: 4 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Rebase are o reputație proastă nemeritată. E o unealtă simplă, cu o singură regulă de siguranță care, dacă e respectată, face istoricul proiectului mult mai ușor de citit.

Ce face rebase

Merge combină două istorii și lasă urma acestei combinări (merge commit). Rebase face altceva: ia commit-urile tale și le rejoacă peste alt punct de plecare, ca și cum ai fi început să lucrezi de acolo.

Înainte:

```
A---B---C---F    (main)
      \
      D---E (feature)
```

După `git rebase main`:

```
A---B---C---F---D'---E'    (feature)
```

Observă apostroful: D' și E' nu sunt aceleași commit-uri. Au alt hash, pentru că au alt părinte. Conținutul e același, identitatea nu. Din asta derivă toate regulile de mai jos.

Cum îl folosești

```
git switch feature/export-facturi
git fetch origin
git rebase origin/main
```

Dacă apar conflicte, le rezolvi la fel ca la merge, dar continui altfel:

```
# editezi fișierele, apoi
git add fisier-rezolvat.php
git rebase --continue
```

```
# sau, dacă vrei să renunți complet
git rebase --abort
```

La rebase, conflictele pot apărea de mai multe ori: o dată pentru fiecare commit rejucat. E normal și e motivul pentru care rebase pe un branch cu 30 de commit-uri e obositor.

Regula de aur

Nu faci rebase la commit-uri pe care le-au luat deja alții.

Dacă rescrii istoricul unui branch pe care lucrează și un coleg, la următorul lui pull Git nu mai recunoaște nimic, iar el ajunge cu commit-uri duplicate și o dimineață pierdută.

Practic: rebase pe branch-ul tău, care e doar al tău. Merge pentru ce e public (main, develop, branch-uri partajate).

Push după rebase

Dacă branch-ul era deja publicat, după rebase push-ul normal va fi respins — pe bună dreptate, pentru că istoricul e diferit. Soluția corectă:

```
git push --force-with-lease
```

Folosește `--force-with-lease`, nu `--force`. Diferența e importantă: `--force-with-lease` refuză să suprascrie dacă între timp altcineva a împins ceva ce tu nu ai văzut. `--force` șterge fără să întrebe.

Rebase interactiv: curățenie înainte de review

Asta e partea cu adevărat utilă zilnic. Ai un branch cu commit-uri de lucru:

```
* 9f1a2b3 fix typo
* 3c4d5e6 inca o incercare
* 7a8b9c0 wip
* 1d2e3f4 Adaugă export PDF pentru facturi
```

Nimeni nu vrea să vadă asta într-un pull request. Le combină:

```
git rebase -i HEAD~4
```

Se deschide editorul:

```
pick 1d2e3f4 Adaugă export PDF pentru facturi
pick 7a8b9c0 wip
pick 3c4d5e6 inca o incercare
pick 9f1a2b3 fix typo
```

Schimbă pick în squash (sau s) pe cele care trebuie topite în cel de deasupra:

```
pick 1d2e3f4 Adaugă export PDF pentru facturi
squash 7a8b9c0 wip
squash 3c4d5e6 inca o incercare
squash 9f1a2b3 fix typo
```

Salvezi, Git îți cere un mesaj final și rămâi cu un singur commit curat.

Alte comenzi disponibile în lista aceea: `reword` (schimbă doar mesajul), `edit` (te oprește ca să modifice commit-ul), `drop` (îl elimini), `fixup` (ca squash, dar aruncă mesajul).

Corectarea ultimului commit

Pentru cazul banal — ai uitat un fișier sau ai greșit mesajul — nu ai nevoie de rebase:

```
git add fisierul-uitat.php
git commit --amend --no-edit # păstrează mesajul
git commit --amend -m "Mesaj corect"
```

Și `--amend` rescrie istoricul, deci aceeași regulă: doar pe commit-uri nepublicate.

Cum arată istoricul, în practică

Diferența dintre cele două abordări se vede cel mai bine pe un proiect real, după câteva săptămâni.

Cu merge la fiecare sincronizare, istoricul lui main arată așa:

```
* Merge branch 'main' into feature/facturi
|\
| * Repară afișarea totalului
* | Adaugă export CSV
|/
* Merge branch 'main' into feature/facturi
|\
| * Actualizează dependențele
```

Jumătate din linii nu spun nimic despre produs; sunt doar urme ale sincronizărilor. Cu rebase înainte de integrare, aceeași muncă arată așa:

```
* Adaugă export CSV pentru facturi
* Repară afișarea totalului
* Actualizează dependențele
```

Al doilea istoric poate fi citit de cineva care caută când s-a introdus un bug. Primul, nu prea.

Un rebase tipic, pas cu pas

```
git switch feature/facturi
git fetch origin
git rebase origin/main
```

```
Auto-merging app/Services/Factura.php
CONFLICT (content): Merge conflict in app/Services/Factura.php
error: could not apply 1d2e3f4... Adaugă export CSV
```

Deschizi fișierul, rezolvi exact ca la merge, apoi:

```
git add app/Services/Factura.php
git rebase --continue
```

Dacă apare din nou conflict, repeți. Când termini, Git îți spune Successfully rebased and updated refs/heads/feature/facturi.

Dacă la un pas constăți că un commit a devenit inutil (modificarea lui e deja în main), îl poți sări:

```
git rebase --skip
```

Plasa de siguranță înainte de rebase

Dacă rebase-ul te sperie, fă-ți o etichetă temporară. Costă două secunde și îți garantează întoarcerea:

```
git branch backup/facturi-inainte-de-rebase
git rebase origin/main
# dacă iese prost:
git reset --hard backup/facturi-inainte-de-rebase
```

Aceași siguranță o ai și din git reflog, dar un branch de rezervă e mai ușor de găsit când ești sub presiune.

Merge sau rebase? Răspunsul practic

- Actualizezi branch-ul tău cu ce e nou în main → rebase, ca să nu umpli istoricul de merge commit-uri inutile.
- Integrezi o funcționalitate terminată în main → merge (de obicei prin pull request).
- Cureți commit-urile de lucru înainte de review → rebase interactiv cu squash.
- Branch-ul e partajat cu altcineva → merge, niciodată rebase.

Exercițiu

- Creează un branch cu patru commit-uri, dintre care trei cu mesaje inutile („wip”, „fix”, „iar”).
- Combină-le într-unul singur cu git rebase -i HEAD~4 și scrie un mesaj corect.
- Între timp, adaugă un commit pe main, apoi rulează git rebase main în branch și observă hash-urile noi.
- Explică-ți, cu propriile cuvinte, de ce push-ul ar fi respins dacă branch-ul era deja publicat.

Aplicație practică

Sarcina

Compară merge cu rebase într-un repository de laborator.

Rezolvare orientativă și verificare

Desenează istoricul înainte și după. Folosește rebase doar pe istoricul pentru care poți coordona rescrierea.

Greșeală de evitat

Nu rescrie istoricul partajat fără acordul echipei.

Documentație și surse

- Pro Git — Git

Pull requests pe GitHub

Lectură estimată: 5 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Pull request-ul (PR) nu e o funcție a Git-ului, ci a platformelor de găzduire. Este locul unde munca ta devine vizibilă, e discutată și abia apoi intră în main. În echipe, PR-ul e cel mai important obicei de calitate — mai important decât orice comandă din acest curs.

Fluxul complet

- Pleci de la un main actualizat și deschizi un branch.
- Lucrezi, faci commit-uri, dai push branch-ului.
- Deschizi un pull request de la branch-ul tău către main.
- Colegii comentează; tu faci commit-uri noi în același branch, care apar automat în PR.
- După aprobare, PR-ul e integrat și branch-ul se șterge.

```
git switch main && git pull
git switch -c fix/eroare-la-salvarea-comenzii
# ... modificări, commit ...
git push -u origin fix/eroare-la-salvarea-comenzii
```

După push, Git îți afișează chiar în terminal un link direct de creare a PR-ului.

Cum scrii un pull request pe care lumea vrea să-l revizuiască

Un PR bun economisește timpul reviewerului. Structura care funcționează:

- Titlu — ce face schimbarea, nu pe ce ai lucrat: „Repară eroarea 500 la salvarea comenzii cu cupon expirat”.
- Context — ce problemă rezolvă, cu link către tichet dacă există.
- Ce am schimbat — trei-cinci puncte, nu lista fișierelor (aceea se vede oricum).
- Cum se testează — pașii concreți pentru cineva care nu a atins codul.
- Capturi de ecran — obligatorii pentru orice schimbare vizuală.
- Riscuri — ce ar putea să se rupă, ce trebuie urmărit după deploy.

Dacă scrii în descriere Closes #42, GitHub închide automat tichetul 42 când PR-ul este integrat.

Mărimea contează

Un PR de 2.000 de linii nu primește review, primește „LGTM” (looks good to me) fără ca nimeni să se fi uitat serios. Un PR de 200 de linii primește observații reale.

Dacă o funcționalitate e mare, împarte-o: întâi migrarea de bază de date, apoi logica, apoi interfața. Fiecare bucată trebuie să funcționeze singură.

Cum răspunzi la review

Reviewerul comentează pe linii. Tu ai trei reacții legitime: accepți și modifici, explici de ce ai făcut altfel, sau propui o a treia variantă. Ce nu faci: iei comentariul personal.

```
# faci modificările cerute în același branch
git add .
git commit -m "Extrage validarea cuponului în clasă separată"
git push
```

PR-ul se actualizează singur. Nu deschizi un PR nou pentru corecturi.

Cele trei moduri de integrare

La butonul de merge, GitHub îți dă trei opțiuni, cu efecte diferite asupra istoricului:

- Merge commit — păstrează toate commit-urile tale plus un commit de merge. Istoric complet, dar stufos.
- Squash and merge — topește tot PR-ul într-un singur commit pe main. Cea mai populară alegere: main rămâne curat, un commit = o funcționalitate.
- Rebase and merge — pune commit-urile tale la capătul lui main, fără merge commit. Istoric liniar, dar pierzi gruparea pe PR.

Dacă echipa nu are o preferință, squash and merge e alegerea sigură.

Protejarea branch-ului main

Într-un repository serios, main este protejat din Settings → Branches. Setările care merită activate:

- interzice push-ul direct în main;
- cere cel puțin o aprobare înainte de merge;
- cere ca testele automate (CI) să treacă;
- cere ca branch-ul să fie la zi cu main înainte de integrare.

Astfel, regula „nu lucrăm direct pe main” nu mai depinde de disciplina fiecăruia.

Cum faci tu un review bun

Mai devreme sau mai târziu vei fi de cealaltă parte. Un review util respectă câteva reguli simple:

- Separă obligatoriul de preferință. Marchează clar: „blocant” pentru ce trebuie schimbat, „sugestie” pentru ce ar fi frumos. Altfel autorul nu știe ce e negociabil.
- Comentează codul, nu omul. „Aici se poate face o injecție SQL” e util. „Nu înțelegi cum funcționează” nu e.
- Întreabă când nu înțelegi în loc să presupui că e greșit. Adesea răspunsul e un context pe care nu-l aveai.
- Verifică ce lipsește, nu doar ce e scris: tratarea erorilor, cazurile limită, testele, ce se întâmplă la valori goale.
- Aprobă când e suficient de bun. Un PR nu trebuie să fie perfect, trebuie să fie mai bun decât ce era.

Pe GitHub ai trei verdicte: Comment (observații fără decizie), Approve și Request changes. Ultimul blochează integrarea, deci folosește-l pentru probleme reale, nu pentru spații.

Automatizări care fac review-ul mai ușor

O parte din comentariile clasice („ai uitat un punct și virgulă”, „formatarea nu respectă standardul”) nu ar trebui făcute de oameni. Le poți muta în GitHub Actions, adică într-un fișier `.github/workflows/ci.yml` care rulează la fiecare push:

```
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: shivammathur/setup-php@v2
        with:
          php-version: '8.3'
      - run: composer install --no-interaction
      - run: php artisan test
```

Rezultatul apare direct în PR, cu bifă verde sau roșie. Combinat cu regula „nu se face merge dacă testele pică”, ai o plasă de siguranță care nu depinde de atenția nimănui.

Contribuția la proiecte open source

La proiectele unde nu ai drept de scriere, fluxul are un pas în plus: faci fork (o copie a repository-ului în contul tău), lucrezi acolo și deschizi PR dinspre fork către proiectul original.

```
git clone https://github.com/contul-tau/proiect.git
cd proiect
git remote add upstream https://github.com/proiect-original/proiect.git
```

```
# ca să te menții la zi cu proiectul original
git fetch upstream
git merge upstream/main
```

Înainte să deschizi PR-ul, citește CONTRIBUTING.md. Multe PR-uri sunt respinse nu pentru cod, ci pentru că nu au respectat convențiile casei.

Exercițiu

- Creează un repository pe GitHub și activează protecția pe main (cere PR înainte de merge).
- Deschide un branch, fă o modificare mică și creează un PR cu descriere completă: context, ce ai schimbat, cum se testează.
- Lasă-ți singur un comentariu de review pe o linie, apoi rezolvă-l cu un commit nou.
- Integrează PR-ul cu „Squash and merge” și verifică ce a apărut în istoricul lui main.

Aplicație practică

Sarcina

Pregătește un pull request cu problemă, schimbare și verificare.

Rezolvare orientativă și verificare

Include un exemplu înainte/după și rezultatele testelor. Diferența trebuie să rămână concentrată pe o singură problemă.

Greșeală de evitat

O descriere lungă nu compensează o modificare greu de verificat.

Documentație și surse

- Pro Git — Git

Recuperarea din greșeli

Lectură estimată: 5 minute

Estimare pentru text, la un ritm de 180 de cuvinte pe minut. Timpul pentru analizarea codului, exerciții și proiecte se adaugă separat și depinde de ritmul tău.

Lecția asta e cea la care te vei întoarce. Nu pentru că vei greși des, ci pentru că, atunci când se întâmplă, ai nevoie de răspunsul corect în treizeci de secunde, nu de o oră de căutat pe forumuri.

Principiul care te liniștește: în Git, aproape nimic din ce a fost commis nu se pierde definitiv. Chiar și după comenzi care par distructive, commit-urile rămân în baza de date locală zile întregi.

1. Am modificat un fișier și vreau să renunț

```
# un fișier
git restore src/config.js

# tot
git restore .
```

Atenție: modificările nesalvate se pierd definitiv. Git nu le-a văzut niciodată, deci nu le poate recupera.

2. Am dat git add din greșeală

```
git restore --staged fisier.txt
```

Scoate fișierul din staging, dar păstrează modificările în folder. Varianta veche, pe care o vei vedea în tutoriale, e `git reset HEAD fisier.txt` — face același lucru.

3. Am greșit mesajul ultimului commit

```
git commit --amend -m "Mesajul corect"
```

Doar dacă nu ai dat încă push. Dacă ai dat, vei avea nevoie de `--force-with-lease` și de acordul echipei.

4. Am uitat un fișier din ultimul commit

```
git add fisierul-uitat.php
git commit --amend --no-edit
```

5. Vreau să anulez ultimul commit, dar să păstrez modificările

```
git reset --soft HEAD~1
```

Commit-ul dispare, modificările rămân în staging, gata de recommit. Util când ai commis prea devreme sau ai amestecat două schimbări.

6. Vreau să șterg ultimul commit cu tot cu modificări

```
git reset --hard HEAD~1
```

Aici chiar pierzi munca necommisă. Înainte de orice `--hard`, obișnuiește-te să rulezi `git status` și, la nevoie, `git stash -u`.

Cele trei feluri de reset

```
git reset --soft HEAD~1 # șterge commit-ul, păstrează modificările în staging
git reset --mixed HEAD~1 # șterge commit-ul, păstrează modificările nestaged (implicit)
git reset --hard HEAD~1 # șterge commit-ul și modificările
```

7. Commit-ul e deja publicat și trebuie anulat

Aici nu folosești reset. Istoricul public nu se rescrie. Folosești revert, care creează un commit nou ce anulează efectul celui vechi:

```
git revert 8f3a1c2
```

E varianta corectă și sigură în producție: istoricul rămâne intact și se vede clar că a existat o problemă și că a fost anulată.

8. Am șters un branch din greșeală

Aici intervine eroul lecției: git reflog. El înregistrează toate mișcările lui HEAD, inclusiv cele „pierdute”.

```
git reflog
```

```
9c1d2e3 HEAD@{0}: checkout: moving from feature/export to main
4b7e9a1 HEAD@{1}: commit: Adaugă generarea PDF
8f3a1c2 HEAD@{2}: commit: Structura inițială a exportului
```

Vezi hash-ul ultimului commit din branch-ul șters și îl recreezi:

```
git branch feature/export-recuperat 4b7e9a1
```

9. Am făcut reset --hard și vreau înapoi

Același reflog:

```
git reflog
git reset --hard HEAD@{1}
```

Atâta vreme ce commit-urile au existat, sunt recuperabile aproximativ 90 de zile, până la curățarea automată.

10. Am făcut push cu ceva ce nu trebuia

Dacă e vorba de cod greșit: git revert și push normal.

Dacă e vorba de un secret (parolă, cheie API): prima acțiune e să schimbi secretul, nu să cureți istoricul. Abia apoi cureți, cu git filter-repo sau BFG Repo-Cleaner, și anunți echipa că istoricul a fost rescris.

11. Nu știu ce se întâmplă, vreau starea curată de pe server

```
git fetch origin
git reset --hard origin/main
git clean -fd # șterge fișierele untracked
```

Comanda asta aruncă tot ce ai local. Rulează întâi git status și git stash -u dacă ai vreun dubiu.

Cum afli ce commit a stricat ceva

Când o funcționalitate mergea acum o lună și azi nu mai merge:

```
git bisect start
```

```
git bisect bad          # starea curentă e stricată
git bisect good 8f3alc2  # aici mergea
# Git te plimbă prin commit-uri; la fiecare testezi și spui:
git bisect good        # sau: git bisect bad
git bisect reset
```

Prin căutare binară, găsești commit-ul vinovat în câțiva pași chiar și printre sute.

Util și pentru „cine a scris linia asta și de ce”:

```
git blame -L 40,60 app/Services/FacturaService.php
```

Tabel de urgență

Renunț la modificări nesalvate	→ git restore .
Scot din staging	→ git restore --staged fisier
Corectez ultimul commit	→ git commit --amend
Anulez commit, păstrez modificări	→ git reset --soft HEAD~1
Anulez commit publicat	→ git revert <hash>
Recuperez orice	→ git reflog
Pun deoparte temporar	→ git stash -u
Aflu ce a stricat	→ git bisect / git blame

Exercițiu

- Fă un commit, apoi git reset --hard HEAD~1. Recuperează-l folosind git reflog.
- Creează un branch, fă două commit-uri, șterge branch-ul cu -D și recreează-l din reflog.
- Fă un commit cu o greșeală evidentă, dă push, apoi anulează-l corect cu git revert.
- Scrie-ți propriul tabel de urgență și ține-l la îndemână.

Ai terminat cursul

Știi acum fluxul complet: repository, commit-uri curate, branch-uri, merge, rezolvarea conflictelor, rebase, pull requests și recuperarea din greșeli. Restul vine din repetiție — folosește Git pentru orice proiect, inclusiv notițe sau fișiere de configurare. În două săptămâni comenzile devin reflex.

Aplicație practică

Sarcina

Recuperează un commit folosind reflog într-un repository fictiv.

Rezolvare orientativă și verificare

Identifică hash-ul și creează un branch nou care îl păstrează. Pentru anularea publică preferă revert când este potrivit.

Greșeală de evitat

Nu experimenta cu reset --hard pe munca nesalvată.

Documentație și surse

- Pro Git — Git

Atelier final: Conflict de Git cu recuperare

Timp de lucru estimat: 40 de minute. Folosește datele de laborator de mai jos și salvează rezultatul separat de orice proiect real.

Date de pornire

Repository de laborator cu README. Branchul main schimbă instrucțiunea de instalare, branchul feature schimbă aceeași linie și adaugă un exemplu. Nu folosi un repository de producție.

1. Pregătire — 5 minute

Extrage intrările și condițiile din enunț. Notează ce rezultat trebuie produs și ce informații nu sunt confirmate. Pregătește fișierele sau tabelul în care vei păstra dovada lucrului.

2. Construire — 15 minute

Creează conflictul, citește ambele intenții și păstrează instrucțiunea corectă plus exemplul. Înregistrează commiturile, apoi exersează recuperarea unui commit pe un branch separat folosind reflog.

3. Variație și verificare — 15 minute

Prezintă diff-ul ca într-un pull request: problemă, soluție și verificare. Arată ce fișiere temporare trebuie excluse înainte de commit.

4. Autoevaluare — 5 minute

Compară rezultatul cu reperele de mai jos. Înregistrează cel puțin o eroare identificată sau un caz-limită verificat, plus corectura ori explicația comportamentului observat.

Repere de rezolvare

Conflictul este rezolvat fără marcaje rămase și fără pierderea exemplului. Commitul recuperat este referit de un branch. Istoricul partajat nu este rescris pentru a simplifica exercițiul.

Ce predai

- Repository
- Branch
- Conflict rezolvat
- Pull request
- Exercițiu reflog

Criterii de acceptare

- Nu sunt secrete în commit
- Diff-ul este revizuit
- Commitul recuperat este păstrat pe un branch

Păstrează o versiune inițială și una revizuită, cu explicația diferențelor. Dacă un criteriu nu este îndeplinit, revino la lecția relevantă și repetă verificarea. Durata este orientativă; instalarea mediului și aprofundarea pot necesita timp suplimentar.