Git

Distribuirani sistem za kontrolu verzije

Šta je Git?

Git je sistem za kontrolu verzije koji omogućava praćenje i upravljanje promenama u kodu tokom razvoja softvera. Iako je inicijalno razvijen sa idejom podrške koordinaciji tima programera koji zajedno rade na određenom projektu, može biti korišćen i u generalne svrhe, za praćenje promena u okviru proizvoljne kolekcije datoteka.

Git se posebno ističe po tome što podržava distribuirani razvoj, što znači da ne postoji *centralni* repozitorijum. Svaka osoba ima kompletnu kopiju repozitorijuma sa celom istorijom na svom računaru, što omogućava rad bez stalne internet konekcije.

Međutim, u praksi, većina timova i projekata uspostavlja jedan repozitorijum koji služi kao zajednička referentna tačka – ono što predstavlja *centralni* repozitorijum. Ovo je više organizaciona konvencija nego tehnička nužnost.

Platforme poput GitHub-a, GitLab-a ili Bitbucket-a se koriste za deljenje/skladištenje repozitorijuma i samo zbog dogovora tima postaju glavna tačka koordinacije.

Koje probleme Git rešava?

Git rešava problem praćenja promena u kodu i omogućava timovima da efikasno sarađuju na projektima. Bez sistema za kontrolu verzije, bilo bi teško pratiti ko je napravio koje izmene, kada su te izmene napravljene i zašto.

- **Kontrola verzije**: Nema standardnog sistema za označavanje verzija
- **Dupliranje podataka**: Svaka verzija je potpuna kopija celog projekta
- **Nemogućnost grananja**: Teško je raditi paralelno na različitim funkcionalnostima
- **Problem integracije - timovi**: Spajanje DEV2 promena je manuelni proces
- **Nedostatak metapodataka**: Nema informacija o tome ko je, šta, kada i zašto promenio

```
projekat/
├── shared/
│   └── node_modules/ # Jedna kopi
├── v1-final/
│   ├── node_modules -> ../shared/
│   ├── index.html
│   └── style.css
├── v2-final/
│   ├── node_modules -> ../shared/
│   ├── index.html
│   └── style.css
├── v2-FINAL-STVARNO/
│   ├── node_modules -> ../shared/
│   ├── index.html
│   └── style.css
├── v2-FINAL-STVARNO-POPRAVKA/
│   ├── node_modules -> ../shared/
│   ├── index.html
│   └── style.css
└── DEV2-verzija-novo/
    ├── node_modules -> ../shared/
    ├── index.html
    └── style.css
```

Kako je Git nastao?

Git je razvijen 2005. godine od strane Linusa Torvaldsa, kreatora Linux operativnog sistema, kao odgovor na potrebu za boljim sistemom kontrole verzije koji bi mogao da podrži razvoj Linux kernela. Pre Git-a, Linus je koristio BitKeeper, ali je odlučio da razvije sopstveni sistem kada su se pojavili problemi sa licenciranjem i dostupnošću BitKeeper-a. Git je brzo postao popularan zbog svoje brzine, efikasnosti i sposobnosti da podrži distribuirani razvoj.

Linus Torvalds 2005.
- Napravio Git za 5 dana bez SO, AI
- Piše C kod direktno u Vimu-u bez autocomplete-a
- Napisao i održava Linux kernel

Developer 2025.
- Pita AI da mu napravi tekst za mim
- Kopira ceo kod od AI bez razumevanja
- Deli projekat klijentu preko localhost linka

Gde sve počinje: `git init`

Da bismo uopšte počeli da korisitmo Git, prvi korak je da mu kažemo *gde* da radi. To radimo **inicijalizacijom repozitorijuma**, odnosno komandom `git init`. Komanda `git init` transformiše običan direktorijum u Git repozitorijum. Ova komanda kreira skriveni `.git` poddirektorijum. Tu Git čuva sve svoje podatke.

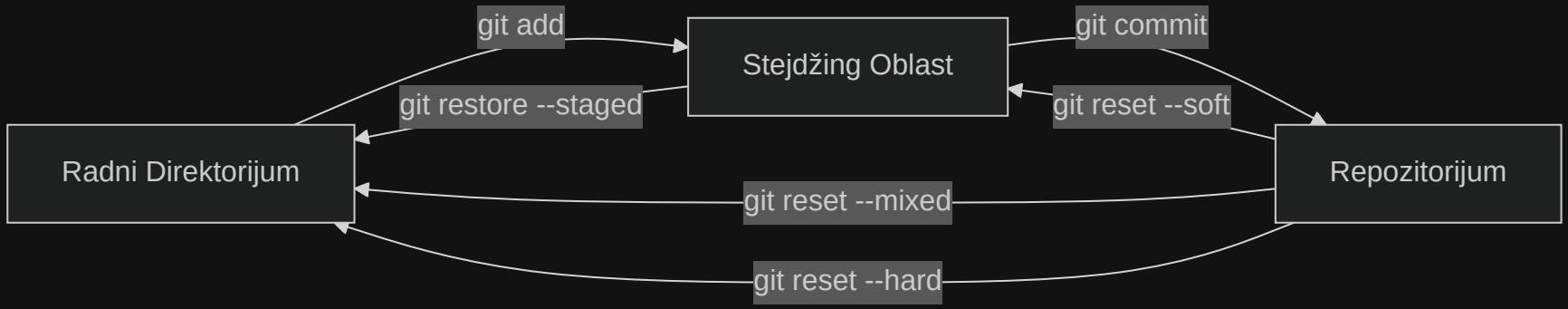
```
➜  git-presentation git init
Initialized empty Git repository in /home/dragomirm/Projects/
➜  git-presentation git:(main) ✗ tree -L 2 .git
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

9 directories
```

Tri ključna stanja/prostora

Nakon `git init`, Git posmatra fajlove kroz tri osnovna stanja ili prostora:

- **Radni Direktorijum** (Working Directory) - Fajlovi koje vidimo i menjamo
- **Stejdžing Oblast** (Staging Area/Index) - Mesto gde pripremamo promene koje želimo da uključimo u sledeći **snapshot** (commit)
- **Repozitorijum** (Repository) - Trajna baza podataka gde Git čuva sve *snapshot-ove* (commit-ove)



Tri ključna objekta

Kada uradimo `git commit`, Git ne pamti samo razlike (diffs/delta), već kreira **snapshot** radnog direktorijuma vašeg projekta koristeći tri osnovna tipa objekata. Svi se čuvaju u `.git/objects` i imaju jedinstveni SHA-1 heš.

1. Blob

- Čuva **sirovi sadržaj** fajla (samo podatke).
- Ne sadrži ime fajla, putanju ili vreme izmene.
- Heš zavisi samo od sadržaja. Isti sadržaj = isti blob.

2. Tree

Predstavlja **direktorijum**. Sadrži listu pokazivača (heševa) na:

- Blob (fajlovi u direktorijumu)
- Tree (poddirektorijumi)

Pamti i ime i mod (npr. izvršni) za svaki unos. Kao "manifest" direktorijuma.

3. Commit

Predstavlja **jedan snapshot** projekta. Sadrži:

- Pokazivač na **Tree**
- Pokazivač(e) na **roditeljski(e) commit(ove)**
- Metapodatke (autor, vreme, poruka...)

DEMO 1

(init, prostori/stanja, objekti, itd.)

Grananje

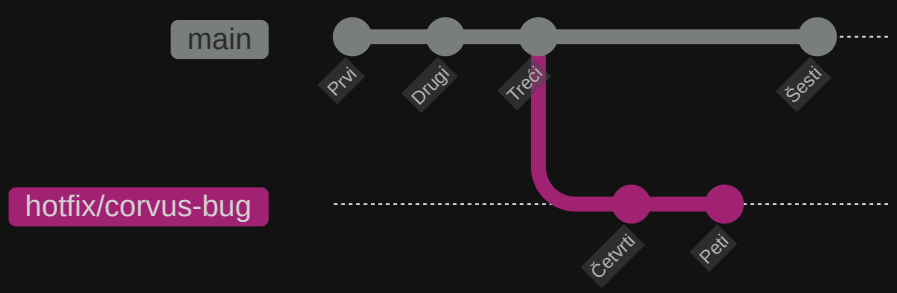
Umesto linearnog rada gde svaka promena odmah utiče na glavnu liniju razvoja, grananje nam omogućava da kreiramo izolovane linije rada, što donosi brojne prednosti, posebno u timskom okruženju ili pri radu na kompleksnim projektima.

- **Izolacija rada:** Razvijajte nove funkcionalnosti (features) ili ispravljajte greške (bug fixes/hotfix) bez ometanja stabilne, produkcione verzije koda (obično `main` grana).
- **Eksperimentisanje bez rizika:** Isprobajte nove ideje, refaktorisanje ili implementacije na posebnoj grani. Ako ne uspe, jednostavno je obrišite bez posledica po glavni kod.
- **Paralelni razvoj:** Omogućava da više developera ili timova rade na različitim delovima projekta istovremeno, svako na svojoj grani.
- **Bolja organizacija:** Projekat postaje pregledniji jer svaka grana može predstavljati specifičan zadatak, funkcionalnost ili verziju (npr. `feature/user-login`, `hotfix/decija-sela-ponovo`).
- **Priprema za izdanje (release):** Koristite posebne grane za pripremu nove verzije softvera, testiranje i stabilizaciju pre objavljivanja (npr. `dev`, `stg`, `prod`).

Šta je grana?

Iako intuitivno zamišljamo granu kao kompletnu kopiju projekta, u Gitu je ona mnogo jednostavnija i efikasnija. Grana nije ništa drugo nego lagani, pokretni pokazivač (referenca/refs/pointer) koji ukazuje na jedan specifičan commit u istoriji projekta. Git interno upravlja ovim pokazivačima koristeći jednostavne fajlove unutar `.git/refs/` direktorijuma.

- **Pokazivač na commit:** Osnovna uloga grane je da "zapamti" određeni commit
- **Interno skladištenje:** Svaka grana se čuva kao fajl unutar `.git/refs/heads/`. Ime fajla odgovara imenu grane (npr. `.git/refs/heads/main`).
- **Sadržaj fajla:** Unutar tog fajla nalazi se samo 40-karakterni `SHA-1` heš commit-a na koji grana trenutno pokazuje.
- **Efikasnost:** Kreiranje nove grane je izuzetno brzo jer Git samo kreira novi fajl sa postojećim `SHA-1` hešom. Commit-ovanje na grani samo ažurira `SHA-1` heš u tom fajlu.



Gde nam je glava?

Da bi znao na kojoj se verziji koda trenutno nalazite i gde će sledeći commit biti dodat, Git koristi specijalni pokazivač pod nazivom `HEAD`. On služi kao referenca na vašu trenutnu lokaciju u istoriji projekta i određuje koje stanje fajlova se nalazi u vašem radnom direktorijumu.

- **Uloga `HEAD`:** Pokazuje na commit koji predstavlja trenutno stanje vašeg radnog direktorijuma.
- **Normalno stanje (vezan `HEAD`):** Najčešće, `HEAD` pokazuje na ime neke grane. To znači da ste "na toj grani".
- **Interno skladištenje:** Lokacija `HEAD`-a se čuva u fajlu `.git/HEAD`.
- **Sadržaj `.git/HEAD` (vezan):** Kada je `HEAD` vezan za granu (npr. `main`), fajl sadrži simboličku referencu: `ref: refs/heads/main`
- **Promena `HEAD`-a:** Komande kao `git switch` ili `git checkout grana` menjaju sadržaj `.git/HEAD` da pokazuje na novu granu i ažuriraju radni direktorijum.

```
➜  git-presentation git:(main) ✗ cat .git/HEAD
ref: refs/heads/main
```

Posebna stanje (Detached HEAD)

Ponekad želimo da pogledamo ili radimo sa stanjem projekta koje ne odgovara vrhu nijedne postojeće grane, već nekom specifičnom commit-u iz prošlosti. Git ovo omogućava kroz stanje poznato kao "Detached HEAD", gde `HEAD` pokazivač nije vezan za ime grane, već direktno za `SHA-1` heš commit-a.

- **Ulazak u stanje:** Dešava se kada uradite `checkout` na SHA-1 heš commit-a ili na tag (koji pokazuje na commit):
- **Stanje `HEAD`-a:** `HEAD` više ne pokazuje na granu, već direktno na commit.
- **Sadržaj `.git/HEAD` (odvezan):** Fajl sada sadrži direktno SHA-1 heš commit-a:
  `a4f12e8ab76c9e7f5d3a8f1e8ddc2921b1a8a55f`
- **Korisnost:** Idealno za pregledanje starog koda, buildovanje specifične verzije, ili pokretanje testova na istorijskom commit-u.
- **Važno:** Novi commit-ovi napravljeni u ovom stanju ne pripadaju nijednoj grani. Ako se prebacite na drugu granu bez kreiranja nove iz ovog stanja, ti commit-ovi mogu biti izgubljeni (očišćeni od strane Git-a).
- **Rešenje:** `git switch -c nova-grana-odavde` pre prebacivanja.

Osnovne komande za rad sa granama

Za efikasno korišćenje sistema grananja, Git nudi nekoliko osnovnih komandi koje omogućavaju kreiranje novih grana, prelazak između postojećih, kao i njihovo listanje i brisanje.

Izlistavanje grana

Prikazuje sve lokalne grane. Zvezdica (`*`) označava trenutnu granu (gde je `HEAD`).

```
git-presentation git:(master) ✗ git branch
```

```
bugfix/corvus
feature/facer-app-adjustments
* master
```

Da bismo prikazali i udaljenje (remote) grane, koristimo parametar `-a` ili `--all` .

```
git-presentation git:(master) ✗ git branch --all
```

Osnovne komande za rad sa granama 2

Kreiranje nove grane

Kreira novu granu koja pokazuje na isti commit kao i trenutna grana (`HEAD`), ali ne prelazi na nju.

```
git-presentation git:(main) ✗ git branch ime-nove-grane
```

Prelazak na drugu granu

Menja `HEAD` da pokazuje na specificiranu granu i ažurira radni direktorijum.

```
git-presentation git:(main) ✗ git checkout ime-nove-grane
# Modernija alternativa
git-presentation git:(main) ✗ git switch ime-nove-grane
```

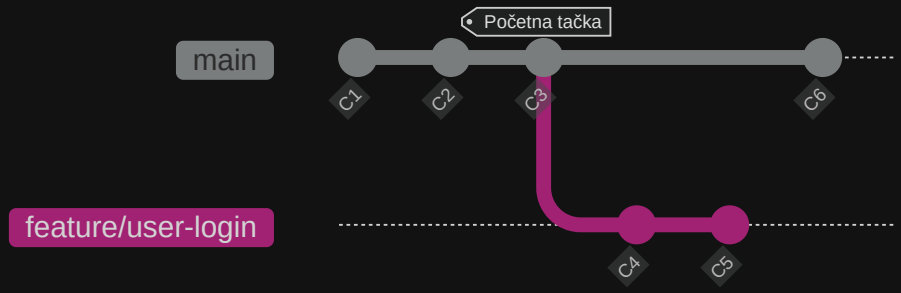
Kreiranje i prelazak odjednom

Skraćenica koja kreira novu granu i odmah prelazi na nju.

Rad na granama

Lepota grananja leži u tome što omogućava da se istorija projekta prirodno razilazi (diverged branches). Kada pređete na novu granu i počnete da pravite nove commit-ove, te promene se beleže samo na toj grani, nezavisno od drugih grana koje možda postoje.

- **Nezavisne istorije:** Commit-ovi napravljeni na jednoj grani ne utiču direktno na druge grane.
- **Pomeranje pokazivača:** Kada napravite commit, Git automatski pomera pokazivač (fajl u `.git/refs/heads/`) **samo za granu na kojoj se trenutno nalazite** (onu na koju pokazuje `HEAD`) na novi commit.
- `HEAD` **prati granu:** Pošto je `HEAD` obično vezan za granu, on se takođe pomera zajedno sa njom na novi commit.
- **Razilaženje:** Ako se commit-ovi prave nezavisno na dve različite grane (npr. `main` i `feature-x`) koje su nekada delile zajedničkog pretka, njihove istorije počinju da se razlikuju.



Spajanje grana (merging): Integracija rada

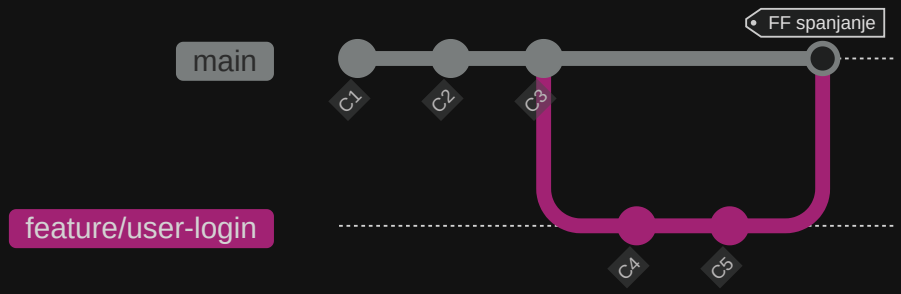
Nakon što završite rad na zasebnoj grani (npr. implementirate novu funkcionalnost ili ispravite grešku), sledeći logičan korak je da te promene integrišete nazad u glavnu liniju razvoja ili neku drugu relevantnu granu (npr. `dev` ili `stg`). Ovaj proces spajanja istorija naziva se `merge`.

- **Cilj spajanja:** Kombinovati promene napravljene na jednoj grani (`source` grana) sa stanjem druge grane (`target` grana).
- **Osnovna komanda:** `git merge <ime-source-grane>`
- **Tipičan tok rada**:
  - Pređite na granu **u koju** želite da spojite promene (target grana): `git switch main`
  - Pokrenite komandu za spajanje, navodeći granu **čije** promene želite da integrišete (source grana): `git merge feature/user-login`
- **Rezultat:** Target grana (`main` u primeru) će sada sadržati sve promene iz svog prethodnog stanja, plus sve promene uvedene na `feature/user-login` grani od trenutka kada su se razdvojile.

Tipovi spajanja grana: Fast-Forward

Najjednostavniji scenario spajanja dešava se kada grana u koju spajate (target) nije imala nikakvih novih commit-ova od trenutka kada je grana koju spajate (source) kreirana iz nje. U ovom slučaju, istorija je i dalje linearna, a Git može izvršiti "brzo premotavanje".

- **Uslov:** Target grana (`main`) nema nove commit-ove koji nisu prisutni u source grani (`feature/user-login`). Drugim rečima, `feature/user-login` je direktni naslednik `main` .
- **Git-ova akcija:** Nema potrebe za kreiranjem novog "merge commit-a". Git jednostavno pomera pokazivač target grane (`main`) unapred da pokazuje na isti commit kao i source grana (`feature/user-login`).
- **Rezultat:** Istorija ostaje linearna, kao da su svi commit-ovi napravljeni direktno na `main` grani. Spajanje je samo pomeranje pokazivača.



Tipovi spajanja: Three-Way Merge

Kada su obe grane (i target i source) imale nezavisne promene (nove commit-ove) od trenutka kada su se razdvojile od zajedničkog commit-a, Git ne može jednostavno da pomeri pokazivač. Umesto toga, mora da izvrši kompleksnije spajanje poznato kao "three-way merge", koje uključuje kreiranje novog commit-a.

- **Uslov:** Obe grane (`main` i `feature/user-login`) imaju nove commit-ove od njihovog poslednjeg zajedničkog commit-a (merge base). Istorija se razilazi.
- **Git-ova akcija**:
  - Pronalazi poslednji zajednički commit (merge base) obe grane.
  - Pokušava da automatski spoji promene napravljene na obe grane od tog pretka.
  - Kreira **novi commit (merge commit)**. Ovaj commit je poseban jer ima **dva roditeljska commit-a**: poslednji commit na target grani i poslednji commit na source grani.
- **Rezultat:** Merge commit integriše obe nezavisne linije razvoja. Istorija sada jasno pokazuje tačku spajanja.

Konflikti pri spajanju (Merge Conflicts)

Iako Git odlično radi automatsko spajanje promena, ponekad naiđe na situaciju gde ne može samostalno da odluči kako da integriše izmene. Ovo se dešava kada su iste linije koda modifikovane na različite načine u obe grane koje se spajaju, što dovodi do konflikta.

- **Uzrok:** Ista sekcija fajla je izmenjena nezavisno na obe grane (`target` i `source`).
- **Git-ova reakcija**:
  - Zaustavlja proces spajanja (`merge`).
  - Označava fajlove koji sadrže konflikte.
  - Ubacuje specijalne markere u konfliktne delove fajla da pokaže obe verzije izmena.
- **Konfliktni Markeri:** Izgledaju ovako u kodu:

```
\<<<<<<< HEAD
// Kod iz TRENUTNE grane (na kojoj ste pokrenuli merge, npr. main
Ovo je kod sa main grane.
=======
// Kod iz GRANE KOJU SPAJATE (npr. feature/user-login) (theirs)
```

Konflikti pri spajanju (Merge Conflicts) 2

- **Rešavanje konflikta (Manuelni proces)**:
  - **Identifikuj konflikte:** `git status` će pokazati fajlove sa konfliktima ("Unmerged paths").
  - **Otvori fajl:** Koristi editor teksta da otvoriš svaki konfliktni fajl.
  - **Odluči i izmeni:** Pregledaj označene delove. Odluči koja verzija koda je ispravna, ili ih kombinuj na željeni način.
  - **Ukloni markere:** Obavezno obriši `<<<<<<<` , `=======` , i `>>>>>>>` linije.
  - **Dodaj rešeni fajl:** `git add <ime-fajla>` da označiš konflikt kao rešen.
  - **Završi spajanje:** Kada su svi konflikti rešeni i fajlovi dodati (`git add`), napravi commit: `git commit`

Brže rešavanje (prihvatanje jedne strane): Ako želite da u potpunosti prihvatite SVE promene iz jednog fajla sa jedne od grana i odbacite sve promene iz druge za taj fajl. **Prihvati "našu" verziju (iz trenutne grane):**

```
git checkout --ours putanja/do/fajla
```

Prihvati "njihovu" verziju (iz grane koja se spaja):

```
git checkout --theirs putanja/do/fajla
```

Čišćenje: Brisanje nepotrebnih grana

Jednom kada je rad na grani završen i njene promene su uspešno spojene (merged) u odgovarajuću glavnu granu (npr. `main` ili `dev`), ta grana često više nije potrebna. Dobra praksa je obrisati takve grane kako bi repozitorijum ostao čist i pregledan.

- **Zašto brisati?** Smanjuje "šum" pri listanju grana (`git branch`), olakšava navigaciju i održava fokus na aktivnim linijama razvoja.
- **Bezbedno brisanje (spojene grane):** Koristi se flag `-d` (ili `--delete`). Git će proveriti da li su sve promene sa te grane spojene u granu na kojoj se trenutno nalazite (`HEAD`). Ako jesu, obrisaće granu. Ako nisu, odbiće brisanje uz upozorenje.

```
git branch -d feature/user-login
```

- **Prisilno brisanje (nespojene grane):** Koristi se flag `-D` (isto kao `--delete --force`). Ova komanda će obrisati granu **bez obzira** da li su njene promene spojene ili ne.

```
git branch -D feature/user-login
```

DEMO 2

(head, refs, grane, spajanje, konflikti, itd.)

Šta je git rebase? (Promena baze)

Pored spajanja (merge), Git nudi još jedan, fundamentalno drugačiji način za integraciju promena iz jedne grane u drugu: rebase. Osnovna ideja rebase-a je da uzme niz commit-ova sa jedne grane i ponovo ih primeni, jedan po jedan, na vrhu druge grane. Ovo efektivno menja "bazu" (početnu tačku, merge base) vaše grane, ali ne predstavlja spajanje u istom smislu kao merge.

- **Koncept:** Umesto da spaja dve istorije kreiranjem merge commit-a, `rebase` "premešta" ili "presađuje" celu granu tako da ona počinje od novijeg commit-a na ciljnoj grani.
- **Linearna istorija:** Glavna prednost (ali i potencijalna opasnost) `rebase` -a je što rezultira čistijom, **linearnom istorijom** projekta, bez dodatnih merge commit-ova koji mogu "zagušiti" log.
- **Promena commit-ova:** Važno je razumeti da `rebase` ne premešta *bukvalno* stare commit-ove. On kreira **nove commit-ove** na ciljnoj grani koji imaju isti sadržaj (patch) i poruku kao originalni, ali dobijaju nove SHA-1] heševe jer imaju drugačijeg roditelja i vreme nastanka. Originalni commit-ovi postaju "siročići/mali betmeni" (ali ostaju u `reflog` -u neko vreme).

`rebase` vs. `merge` (Strategija)

Iako obe komande, `git merge` i `git rebase`, služe za integraciju promena, one NISU zamena jedna za drugu. Rade na fundamentalno različite načine, imaju različite posledice po istoriju projekta i koriste se u različitim scenarijima. Razumevanje ovih razlika je ključno za efikasan i bezbedan rad sa Gitom.

- `git merge` (strategija spajanja)
  - Zabeležiti istoriju tačno onako kako se desila, uključujući paralelni rad.
  - Spaja dve linije razvoja. **Čuva originalnu istoriju** obe grane netaknutom.
  - Kreira **novi merge commit** (osim kod fast-forward) koji eksplicitno pokazuje tačku spajanja. Istorija postaje **nelinarna**, ali verna onome što se dogodilo.
  - **Bezbedan** za deljene grane jer ne menja postojeće commit-ove.
  - Finalno spajanje feature grana u glavne linije (`main` , `dev`), integracija gde je važno sačuvati istorijski
- `git rebase` (strategija presađivanja):
  - Napraviti da istorija izgleda kao da je sav rad rađen sekvencijalno.
  - "Presađuje" commit-ove sa jedne grane na vrh druge. **Prepisuje istoriju** kreiranjem novih commit-ova (novi SHA-1).
  - **Linearna istorija**, bez merge commit-ova. Lakša za praćenje linearno, ali sakriva originalni istoriju.
  - **Opasan** za deljene grane jer menja commit-ove.
  - Ažuriranje *lokalne* feature grane sa promenama sa `main` *pre* spajanja; "čišćenje" *lokalne* istorije commit-ova (`rebase -i`) *pre* deljenja grane.

Važno pravilo tokom korišćenja `rebase`-a

`git rebase` je moćan alat, ali postoji jedno apsolutno ključno pravilo koga se morate pridržavati da biste izbegli haos u timskom radu.

Nikada nemojte raditi `rebase` na commit-ovima koji su već podeljeni (push-ovani) na zajednički repozitorijum!

- **Zašto?** Jer `rebase` **menja postojeće commit-ove** (kreira nove sa drugačijim SHA-1 heševima).
- **Problem:** Ako ste vi push-ovali granu (`feature/user-login`) i onda uradili `rebase` i ponovo je push-ovali (koristeći `--force`), vaša lokalna istorija i istorija na udaljenom repozitorijumu se **razilaze**.
- **Posledice za tim:** Kada drugi članovi tima povuku (`pull`) vašu rebase-ovanu granu, Git će videti dve različite istorije i pokušaće da ih spoji, što dovodi do **dupliranih commit-ova** i izuzetno konfuzne istorije koju je teško ispratiti i popraviti.
- **Force Push:** Da biste uopšte mogli da gurnete rebase-ovanu granu koja je već postojala na remote-u, moraćete da koristite `git push --force` ili `git push --force-with-lease` .
- `--force` : Bezuslovno pregazi udaljenu granu. **Veoma opasno** ako neko drugi radi na njoj!
- `--force-with-lease` : Bezbednija opcija. Pregaziće udaljenu granu samo ako se ona nije menjala (niko drugi nije push-ovao) od vašeg poslednjeg fetch-a. **Preferirana opcija ako baš morate.**

Važno pravilo tokom korišćenja `rebase`-a (izuzetak)

Jedini put kada je u redu raditi rebase na deljenoj grani je ako ste **jedina osoba** koja radi na toj grani ili ste se **eksplicitno dogovorili sa celim timom** i svi znaju kako da rukuju sa rebase-ovanom granom.

Zaključak: Koristite `rebase` slobodno za **sređivanje vaše lokalne istorije *pre* nego što je podelite (push-ujete).** Za integraciju promena *sa* deljenih grana *u* vaše lokalne, `rebase` je često u redu. Za integraciju *vaših* deljenih promena *u* glavne deljene grane, `merge` **je obično bezbedniji izbor.**

Konflikti tokom `rebase`-a

Pošto `rebase` primenjuje commit-ove jedan po jedan na novu bazu, konflikti se mogu javiti tokom ovog procesa. Rešavanje konflikata tokom rebase-a zahteva pažnju, jer se proces može zaustaviti više puta, a značenje `ours` i `theirs` je obrnuto u odnosu na `merge`.

Ukoliko bi došlo do konflikta prilikom `rebase`-a, došlo bi do konflikta i tokom `merge`-a.

Kako nastaju

Konflikt se dešava ako promena u commit-u koji se trenutno primenjuje sa vaše grane (npr. `C5'` sa `feature/user-login`) nije kompatibilna sa trenutnim stanjem koda na novoj bazi (koje uključuje promene sa `main`-a i eventualno prethodno rebase-ovane commit-ove).

- **Git-ova reakcija:**
  - `rebase` proces se **zaustavlja** na commit-u koji je izazvao konflikt.
  - Git vas obaveštava o konfliktu i ostavlja standardne konflikt markere (`<<<<<<<` , `=======` , `>>>>>>>`) u problematičnim fajlovima.
  - `git status` će pokazati fajlove sa konfliktima i da ste usred `rebase` procesa.

Konflikti tokom `rebase`-a 2

Konfliktni markeri i terminologija

```
\<<<<<<< HEAD (ours)
// Kod koji je već bio prisutan na novoj bazi
// (tj. verzija iz grane na koju radite rebase, npr. main)
// Ovo se tokom rebase-a naziva "naša" (ours) verzija.
Ovo je kod sa main grane (ili prethodno rebase-ovanih commit-ova)
=======
// Kod iz COMMIT-A KOJI SE TRENUTNO PRIMENJUJE
// (tj. verzija iz commit-a sa vaše grane, npr. feature/user-logi
// Ovo se tokom rebase-a naziva "njihova" (theirs) verzija.
Ovo je kod iz commit-a koji pravi konflikt sa feature/user-login.
>>>>>>> commit-hash... (theirs)
```

- `ours` **(rebase):** Verzija faila sa **bazne grane** na koju radite rebase (npr. `main`).

Konflikti tokom `rebase`-a 3

Rešavanje konflikta

- **Identifikuj i otvori** fajlove sa konfliktima.
- **Reši konflikte** u editoru (ukloni markere, odluči kako kod treba da izgleda, imajući u vidu obrnuto značenje `ours` / `theirs`).
- **Dodaj rešene fajlove:** `git add /putanja/do/fajla`
- **Nastavi rebase:** Kada su svi konflikti za *taj* commit rešeni i fajlovi dodati, pokrenite: `git rebase --continue` Git će nastaviti sa primenom sledećeg commit-a u nizu.

Brže rešavanje (prihvatanje jedne strane tokom Rebase-a)

Komanda je ista kao i kod `merge`-a, ali, ponovo, **pažljivo sa obrnutim značenjem**.

```
git checkout --theirs putanja/do/fajla
# ili
git checkout --ours putanja/do/fajla
```

Konflikti tokom `rebase`-a 4

Ostale opcije tokom konflikta

- **Preskoči commit:** Ako shvatite da vam commit koji pravi problem više ne treba: `git rebase --skip`
- **Prekini rebase:** Za potpuni povratak na stanje pre početka: `git rebase --abort`
- **Ponavljanje konflikata:** Budite spremni da se isti logički konflikt može pojaviti na više commit-ova tokom `rebase`-a.

Automatsko rešavanje ponovljenih konflikata (Rerere)

- Git ima opciju `rerere` (Reuse Recorded Resolution) koja može biti izuzetno korisna, posebno tokom dugih `rebase` operacija.
- Ako je omogućena (`git config --global rerere.enabled true`), Git će **zapamtiti kako ste ručno rešili** određeni konflikt.
- Ako se **isti konflikt** ponovo pojavi kasnije (npr. na sledećem commit-u tokom istog `rebase`-a), Git će **automatski primeniti vaše prethodno rešenje**, štedeći vam vreme.
- Git će vas obavestiti kada primeni zapamćeno rešenje (npr. `Resolved '<fajl>' using previous resolution.`). I dalje je dobra praksa da proverite rezultat

Interaktivni Rebase: `git rebase -i`

Pored osnovnog "presađivanja" grane, Git nudi i interaktivni rebase (`rebase -i`), izuzetno moćan parametar za preuređivanje, menjanje i kombinovanje commit-ova pre nego što ih podelite sa drugima. Ovo je idealno za "čišćenje" vaše lokalne istorije rada.

- **Svrha:** Srediti niz commit-ova na vašoj lokalnoj grani (npr. `feature/user-login`) pre nego što je push-ujete ili spojite. Cilj je da istorija bude logičnija, čitljivija i da sadrži smislene, atomske commit-ove.
- **Pokretanje:** `git rebase -i <base>` gde je `<base>` commit *pre* prvog commit-a koji želite da menjate.

```
# Primer: Sredi sve commit-ove na feature/user-login
git rebase -i feature/user-login
# Primer: Sredi poslednja 3 commit-a na trenutnoj grani (feature/
git rebase -i HEAD~3
```

Interaktivni Rebase: `git rebase -i` 2

Interaktivni Editor

Git će otvoriti editor teksta sa listom commit-ova koje ste izabrali, obrnutim redosledom (najstariji prvi). Ispred svakog commit-a stoji komanda (podrazumevano `pick`)

```
pick a1b2c3d Fix typo in README
pick e4f5g6h Add basic login functionality # WIP
pick i7j8k9l Implement user logout
pick m0n1o2p Add more detailed logging for login # Temp debug

# Rebase ...
# Commands:
# p, pick <commit> = use commit (Zadrži commit kakav jeste)
# r, reword <commit> = use commit, but edit the commit message (Za
promenite commit poruku)
```

Interaktivni Rebase: `git rebase -i` 3

Interaktivni Editor

Nakon što sačuvate izmene u editoru, Git će pokušati da primeni commit-ove jedan po jedan prema vašim instrukcijama. Ako ste koristili `reword` ili `edit`, zaustaviće se i dati vam priliku da uradite izmene (`git commit --amend`, `git rebase --continue`). Ako dođe do konflikta, rešavate ga kao kod običnog `rebase`-a.

Važno: Interaktivni rebase takođe **menja istoriju** (SHA-1 heševe)! Primenjuje se isto pravilo kao za običan rebase. Koristite ga samo na **lokalnim, nedeljenim** granama.

Promena redosleda: Jednostavno promenite redosled linija u editoru da biste promenili redosled primene commit-ova.

Sigurnosna mreža: `git reflog`

Operacije koje menjaju istoriju, kao što je rebase (ili `git reset --hard` (pokrivamo kasnije)), mogu delovati zastrašujuće. Šta ako slučajno obrišete pogrešan commit ili `rebase` krene naopako? Srećom, Git čuva **privremeni zapis svih promena** na vrhovima vaših grana i `HEAD` -a, poznat kao `reflog` .

- **Šta je `reflog` ?** To je **lokalni** (ne deli se sa drugima) log koji beleži gde su `HEAD` i vrhovi vaših grana bili u prošlosti. Svaki put kada promenite granu, napravite commit, uradite reset, merge ili rebase, Git dodaje zapis u `reflog` .
- **Zašto je koristan?** Omogućava vam da vidite šta ste radili i da se **vratite na prethodna stanja**, čak i ako ti commit-ovi više nisu dostupni preko neke postojeće grane (npr. nakon "neuspelog" rebase-a ili slučajnog `reset --hard`).

```
➜  git-presentation git:(main) ✗ git reflog
a4f12e8 (HEAD -> main) HEAD@{0}: commit: Final touch-ups
98ca1b3 HEAD@{1}: reset: moving to HEAD~1
e4f5g6h HEAD@{2}: rebase -i (finish): returning to refs/heads/mai
e4f5g6h HEAD@{3}: rebase -i (squash): Add login and logout featur
i7j8k9l HEAD@{4}: rebase -i (squash): # Log message for commit #3
```

Sigurnosna mreža: `git reflog` 2

Struktura zapisa

- SHA-1 heš commit-a na koji je `HEAD` pokazivao *posle* operacije.
- Referencu `HEAD@{index}` (npr. `HEAD@{0}` je trenutno stanje, `HEAD@{1}` je prethodno, itd.).
- Akciju koja je dovela do tog stanja (commit, checkout, rebase, reset, merge…).
- **Oporavak stanja:** Ako shvatite da ste napravili grešku (npr. loš rebase ili reset):
  - Pronađite u `git reflog` stanje **pre** problematične operacije. Obratite pažnju na opis akcije i SHA-1 heš.
  - Možete se vratiti na to stanje koristeći `git reset --hard` :

```
# Vrati se na stanje pre poslednje operacije
git reset --hard HEAD@{1}
# Ili, ako znate tačan SHA-1 heš željenog stanja iz reflog-a
git reset --hard b5d8e1f
```

Ili, ako samo želite da kreirate novu granu iz tog starog stanja:

Zaključak: Kada koristiti `merge`, a kada `rebase`?

Izbor između merge i rebase često zavisi od konvencija tima i usvojenog Git workflow-a. Ne postoji univerzalno "tačan" odgovor, ali postoje opšte preporuke koje pomažu u održavanju čiste i razumljive istorije projekta.

- Kada integrišete promene sa **deljene grane** (npr. `main`, `dev`) u vašu **feature granu** (ako želite da sačuvate tačan kontekst spajanja).
- Kada spajate vašu **završenu feature granu** nazad u **glavnu deljenu granu** (npr. `dev` ili `main`). Ovo čuva informaciju o postojanju feature grane i olakšava potencijalni `revert` (pokrivamo posle) celog feature-a ako je potrebno.
- Kada želite da istorija **precizno odražava** šta se i kada dešavalo, uključujući paralelni rad i tačke integracije.
- Kada radite u timu koji **preferira vidljive merge commit-ove** ili nije komforan sa prepisivanjem istorije.
- Za **ažuriranje vaše lokalne feature grane** sa najnovijim promenama sa osnovne deljene grane (`main`, `dev`) **pre** nego što je podelite ili spojite. Ovo rezultira čistijom, linearnom istorijom kada se konačno spoji.
- Za **"čišćenje" vaše lokalne istorije** commit-ova (koristeći `rebase -i`) **pre** nego što podelite granu (push). Možete spojiti WIP commit-ove (`squash`, `fixup`), preformulisati poruke (`reword`), promeniti redosled, itd.
- Kada želite **linearnu, čistu istoriju** bez "šuma" merge commit-ova.
- Kada radite **sami** na grani ili u timu koji je **usvojio rebase workflow** i razume posledice.

DEMO 3

(rebase, interaktivni rebase, konflikti, reflog, itd.)

Poništavanje promena: restore, reset, revert

Greške se dešavaju, planovi se menjaju. Git nudi nekoliko moćnih alata za poništavanje (undo) različitih vrsta promena, od izmena u fajlovima koje još nismo commit-ovali, do poništavanja celih commit-ova koji su već deo istorije. Ključno je razumeti šta rade komande `git restore`, `git reset` i `git revert`, jer operišu na različitim nivoima i imaju različite posledice.

- **Različiti nivoi "Undo":** Ove komande ciljaju različite delove Git-ovog sistema:
  - `git restore`: Fokus na vraćanju stanja **fajlova** u Radnom Direktorijumu (Working Directory) i Stejdžing Oblasti (Staging Area).
  - `git reset`: Menja gde pokazuje **pokazivač grane** (i `HEAD`), utičući i na Stejdžing Oblast i (opciono) na Radni Direktorijum. **Može menjati istoriju.**
  - `git revert`: Poništava promene iz postojećeg commit-a tako što kreira **novi commit** sa suprotnim efektom. **Ne menja postojeću istoriju.**
- **Izbor pravog alata:** Odabir komande zavisi od toga *šta* želite da poništite i da li je ta promena već podeljena sa drugima.

`git restore`: Precizno vraćanje fajlova (novi pristup)

Uvedena u novijim verzijama Gita, komanda git restore pruža jasan i fokusiran način za poništavanje izmena na fajlovima u vašem Radnom Direktorijumu i Stejdžing Oblasti, preuzimajući tu ulogu od ranije preopterećenih komandi `checkout` i `reset`. Njena glavna prednost je što ne pomera `HEAD` niti pokazivač grane.

- **Namena:** Vraćanje fajlova na neko prethodno poznato stanje (npr. stanje poslednjeg commit-a ili stanje u Stejdžing Oblasti).
- **Poništavanje izmena u Radnom Direktorijumu:** Vraća fajl na stanje kakvo je bilo u poslednjem commit-u (`HEAD`).

  Oprez: Ovo briše sve nesnimljene izmene u tom fajlu!

```
# Vrati fajl na stanje poslednjeg commit-a
git restore /putanja/do/fajla
```

- **Uklanjanje fajla iz Stejdžing Oblasti ("Unstaging"):** Vraća fajl u Stejdžing Oblasti na stanje iz `HEAD` -a, efektivno poništavajući `git add` . Izmene u Radnom Direktorijumu ostaju netaknute.

```
# Ukloni fajl iz stejdžinga
git restore --staged /putanja/do/fajla
```

`git restore`: Precizno vraćanje fajlova (novi pristup) 2

- **Vraćanje fajla na stanje iz određenog commit-a:** Možete vratiti fajl na verziju iz bilo kog commit-a, taga ili grane.

```
# Vrati fajl na stanje iz commit-a 'a4f12e8'
git restore --source=a4f12e8 /putanja/do/fajla
# Vrati fajl na stanje sa 'main' grane
git restore --source=main /putanja/do/fajla
```

- **Bezbednost:** Generalno bezbedna komanda za lokalne izmene jer ne dira istoriju commit-ova niti pomera granu. Opasnost leži samo u potencijalnom gubitku **nesnimljenih lokalnih izmena** u Radnom Direktorijumu kada koristite `git restore /putanja/do/fajla`.

- *(Ovo je modernija i jasnija zamena za staro `git checkout HEAD -- /putanja/do/fajla`)*

`git reset`: Pomeranje pokazivača i više

`git reset` je moćna i svestrana komanda koja primarno služi za pomeranje `HEAD`-a i trenutnog pokazivača grane na neki drugi commit. Međutim, zavisno od opcija (`--soft`, `--mixed`, `--hard`), ona takođe može uticati na Stejdžing Oblast i Radni Direktorijum. Ključno je razumeti da `reset` može prepisati istoriju ako pomera granu unazad.

Osnovna upotreba: `git reset [<mod>] <commit>` - Pomera vrh trenutne grane (i `HEAD`) na navedeni `<commit>`.

- `--soft`:
  - **Pomera samo pokazivač grane (i `HEAD`)** na `<commit>`.
  - Stejdžing Oblast i Radni Direktorijum ostaju **netaknuti**.
  - Sve promene iz commit-ova koji su "ostali iza" novog `HEAD`-a sada izgledaju kao "Changes to be committed" (staged).
  - Korisno za brzo "vraćanje" poslednjeg commit-a uz zadržavanje svih izmena u stejdžingu, spremnih za novi, drugačiji commit.
  - Primer: `git reset --soft HEAD~1` (Vrati se za jedan commit, zadrži sve promene u stejdžingu).

`git reset`: Pomeranje pokazivača i više 2

- `--mixed`:
  - **Pomera pokazivač grane** (i `HEAD`) na `<commit>`.
  - **Resetuje Stejdžing Oblast** da odgovara stanju na `<commit>`.
  - Radni Direktorijum ostaje **netaknut**.
  - Sve promene iz commit-ova koji su "ostali iza" sada izgledaju kao "Changes not staged for commit" (untracked/modified).
  - Korisno za poništavanje commit-ova i vraćanje njihovih izmena u radni direktorijum radi daljih modifikacija pre novog commit-a.
  - Primer: `git reset HEAD~1` (Vrati se za jedan commit, izmene su sada unstaged)

`git reset`: Pomeranje pokazivača i više 3

- `--hard`:
  - **Pomera pokazivač grane (i `HEAD`) na `<commit>`.**
  - **Resetuje Stejdžing Oblast** da odgovara stanju na `<commit>`.
  - **Resetuje (prepisuje) Radni Direktorijum** da odgovara stanju na `<commit>`.
  - **Sve promene** iz commit-ova koji su "ostali iza", kao i **sve lokalne izmene** (staged ili unstaged) koje nisu bile u `<commit>` biće **TRAJNO IZGUBLJENE!**
  - Koristiti isključivo kada ste 100% sigurni da želite potpuno da odbacite sve promene posle `<commit>`. Često se koristi uz `reflog` za oporavak.
  - Primer: `git reset --hard a4f12e8` (Vrati sve na stanje commit-a a4f12e8, brišući sve posle njega)

Uticaj na istoriju: Ako `<commit>` na koji resetujete **nije predak** trenutnog `HEAD`-a (tj. ako pomerate granu unazad), `git reset` **menja (prepisuje) istoriju**. Zbog toga važi **pravilo**: Ne radite `reset` (osim `--soft` u nekim slučajevima) na commit-ovima koji su već podeljeni (push-ovani), jer ćete izazvati probleme za druge.

`git revert`: Bezbedno poništavanje commit-ova

Za razliku od `reset`-a koji može prepisati istoriju, `git revert` nudi **bezbedan način** za poništavanje efekata prethodnih commit-ova, posebno onih koji su već podeljeni sa timom. On to radi tako što **ne menja prošlost**, već **kreira novi commit** koji uvodi suprotne promene.

- **Namena:** Poništiti logički efekat jednog ili više postojećih commit-ova.
- **Kako radi:**
  - Navedete commit čije promene želite da poništite: `git revert <commit-hash>`
  - Git izračunava promene potrebne da se poništi efekat tog commit-a.
  - Kreira **potpuno novi commit** koji primenjuje te suprotne promene.
  - Originalni commit ostaje netaknut u istoriji, ali njegov efekat je sada neutralisan novim "revert" commit-om.

```
# Poništi efekat commit-a a4f12e8
git revert a4f12e8
# Git će otvoriti editor za poruku novog revert commit-a
# (Podrazumevana poruka je obično "Revert '<Originalna poruka>'")
# Sačuvajte poruku da biste kreirali revert commit.
```

Poređenje: `restore` vs `reset` vs `revert`

Da sumiramo ključne razlike između ova tri alata za poništavanje promena:

Komanda	Glavni Cilj	Operiše na…	Menja Istoriju?	Bezbednost (Deljene grane)	Tipična Upotreba
`git restore`	Vraćanje stanja **fajlova**	Radni Direktorijum, Stejdžing Oblast	**Ne**	**Bezbedno**	Poništavanje lokalnih izmena u fajlu, unstaging (`restore --staged`)
`git reset`	Pomeranje **pokazivača grane** (i `HEAD`)	Pokazivač grane, `HEAD` , Stejdžing, Radni Dir.	**DA** (često)	**Opasno!** (osim --soft)	Vraćanje grane na stariji commit (lokalno!), čišćenje stejdža/dir (`--hard`)
`git revert`	Poništavanje efekta **commit-a**	Kreira **novi commit**	**Ne**	**Bezbedno**	Poništavanje commit-ova koji su već podeljeni, bez prepisivanja istorije

- **Ključne razlike ukratko:**
  - Želite da poništite **lokalne, necommit-ovane izmene** u fajlu? -> `git restore`
  - Želite da uklonite fajl iz **stejdžinga**? -> `git restore --staged`

DEMO 4

(restore, reset, revert)

Saradnja i udaljeni repozitorijumi (Remotes)

Iako je Git distribuiran sistem gde svako ima kompletnu istoriju, u praksi je za saradnju ili čak samo za lični backup neophodno imati centralnu tačku (ili više njih) gde se kod može deliti i sinhronizovati. Te centralne tačke su **udaljeni repozitorijumi** (remote repositories), a Git nudi mehanizme za efikasnu komunikaciju sa njima.

- **Svrha "Remotes"**
  - Omogućavaju **timsku saradnju** na istom projektu.
  - Služe kao **centralno mesto** za deljenje koda i istorije.
  - Pružaju **backup** vašeg lokalnog repozitorijuma.
  - Omogućavaju rad na projektu sa **više različitih računara**.
- **Koncept veze:** Vaš lokalni Git repozitorijum može imati definisane **veze (references)** ka jednom ili više udaljenih repozitorijuma. Svaka veza ima ime i URL adresu.
- **Komunikacija:** Git koristi ove veze za slanje (`push`) vaših lokalnih promena na remote i za dohvatanje (`fetch` / `pull`) promena koje su drugi napravili na remote-u.

Platforme za hosting: GitHub kao primer

Udaljeni Git repozitorijumi moraju negde da "žive" – na serveru dostupnom preko interneta (ili lokalne mreže). Postoje brojne platforme koje nude uslugu hostovanja Git repozitorijuma, dodajući uz to i niz korisnih alata za upravljanje projektima, praćenje problema, code review, automatizaciju (CI/CD) i još mnogo toga.

- **Popularne platforme:**
  - **GitHub:** Najpopularnija platforma, posebno za open-source projekte, ali i za privatne. Nudi bogat ekosistem alata. Mi ćemo je koristiti kao primer.
  - **GitLab:** Sveobuhvatna DevOps platforma koja nudi Git repozitorijume, CI/CD, praćenje problema, wiki, i može se hostovati i samostalno (self-hosted).
  - **Bitbucket:** Atlassian-ov proizvod, dobro se integriše sa Jira-om i drugim Atlassian alatima, popularan u korporativnom okruženju.
  - Druge: Azure DevOps, SourceForge, AWS CodeCommit, itd.
  -
- **Osnovna funkcionalnost (primer GitHub):**
  - Kreiranje i skladištenje Git repozitorijuma (javnih i privatnih).
  - Web interfejs za pregledanje koda, istorije, grana.
  - Upravljanje pristupom i saradnicima.
  - Mehanizmi za predlaganje promena (Pull Requests).
  - Praćenje problema (Issues).
  - Wiki za dokumentaciju.
  - Automatizacija (GitHub Actions).

`git clone`: Preuzimanje udaljenog repozitorijuma

Najčešći način da počnete da radite na postojećem projektu koji se nalazi na nekoj platformi poput GitHub-a jeste da **klonirate** taj udaljeni repozitorijum. Komanda `git clone` pravi kompletnu lokalnu kopiju repozitorijuma, uključujući svu istoriju i grane.

- **Svrha:** Kreiranje lokalne kopije udaljenog Git repozitorijuma.
- **Komanda:**

```
git clone url-udaljenog-repozitorijuma [opciono-ime-lokalr
```

URL adresu obično dobijate sa platforme (npr. GitHub "Code" dugme), može biti HTTPS ili SSH.

```
# Primer (HTTPS):
git clone https://github.com/popart-studio/decija-sela.git

# Primer (SSH):
git clone git@github.com:popart-studio/decija-sela.git
```

Šta je "Remote"? (`origin`, `upstream`, `git remote`)

Vaš lokalni repozitorijum može biti povezan sa više udaljenih repozitorijuma. Svaka takva veza se naziva **"remote"** i sastoji se lakšu komunikaciju.

- **Listanje postojećih veza (samo imena):**

```
git-presentation git:(main) ✗ git remote
# Izlaz (posle clone-a):
# origin
```

- **Listanje veza sa URL adresama:**

```
git-presentation git:(main) ✗ git remote -v
# Izlaz (posle clone-a):
# origin  https://github.com/popart-studio/... (fetch)
# origin  https://github.com/popart-studio/... (push)
```

- **Uklanjanje ve**

```
git-pre
# Prime
```

- **Promena URL**

```
git-pre
# Prime
```

- **Konvencije za**
  - `origin` : (
  - `upstream`

`git fetch`: Bezbedno dohvatanje promena

Kada želite da vidite šta se novo desilo na udaljenom repozitorijumu (npr. šta su kolege push-ovale na `origin`), ali **niste spremni da odmah spojite** te promene sa vašim lokalnim radom, koristite `git fetch`. Ovo je **bezbedna operacija** koja samo preuzima nove podatke.

- **Svrha:** Preuzeti sve nove podatke (commit-ove, grane, tagove) sa udaljenog repozitorijuma, ali **ne menjati** vaš lokalni radni direktorijum ili vaše lokalne grane.
- **Komanda**:

```
# Dohvati promene sa svih remote veza
git fetch
# Dohvati promene samo sa određenog remote-a (npr. origin)
git fetch origin
# Dohvati promene samo za određenu granu sa remote-a
git fetch origin main
```

- **Šta `fetch` radi**:

`git fetch`: Bezbedno dohvatanje promena 2

Remote-Tracking Branches (npr. `origin/main`)

- Ovo su **lokalne reference** (pokazivači) koje prate stanje grana na udaljenom repozitorijumu u trenutku poslednjeg `fetch`-a (ili `pull`-a/ `clone`-a).
- One se nalaze u `.git/refs/remotes/<ime-remote-a>/<ime-grane>` (npr. `.git/refs/remotes/origin/main`).
- **Ne možete direktno raditi** na ovim granama (ne možete ih `checkout`-ovati i praviti commit-ove na njima). One služe samo kao **ogledalo** stanja remote-a.
- `git fetch origin` će ažurirati gde pokazuje `origin/main`, `origin/dev`, itd.

Posle `fetch`-a: Vaše lokalne grane (`main`, `feature/user-login`) su **netaknute**. Sada možete:

- Videti razlike: `git diff main origin/main`
- Videti log remote grane: `git log origin/main`
- Ručno spojiti promene: `git merge origin/main` (dok ste na `main`)
- Ručno uraditi rebase: `git rebase origin/main` (dok ste na `feature/user-login`)

`git pull`: Dohvatanje i automatsko spajanje

Komanda `git pull` je u suštini prečica koja kombinuje dve operacije: `git fetch` (dohvatanje promena) i odmah zatim `git merge` (ili `git rebase`) za spajanje tih promena u vašu trenutnu lokalnu granu. Iako je zgodna, može biti manje predvidljiva od ručnog `fetch` pa `merge` / `rebase`.

- **Svrha:** Ažurirati vašu trenutnu lokalnu granu sa promenama sa odgovarajuće udaljene grane.
- **Komanda**:

```
# Dohvati promene sa remote-a povezanog sa trenutnom granom # i sp
git pull

# (Dohvati 'main' sa 'origin'-a i spoji u TRENUTNU granu)
git pull origin main
```

- **Podrazumevano ponašanje (`pull = fetch + merge`):**
  - Git izvršava `git fetch` za navedeni remote i granu (ili podrazumevani ako nisu navedeni).
  - Odmah zatim izvršava `git merge` da spoji dohvaćenu granu (npr. `origin/main`) u vašu **trenutnu lokalnu granu**

`git pull`: Dohvatanje i automatsko spajanje 2

Alternativno ponašanje (`pull = fetch + rebase`)

- Možete konfigurisati Git da `pull` koristi `rebase` umesto `merge`-a:

```
# Za trenutni pull
git pull --rebase
# Da bude podrazumevano za ovu granu
git config branch.ime-grane.rebase true
# Da bude podrazumevano globalno (oprez!)
git config --global pull.rebase true
```

- Kada radi `pull --rebase`:
  - Git izvršava `git fetch`.
  - Git "skida" vaše lokalne commit-ove koji ne postoje na remote grani.

`git push`: Deljenje vaših lokalnih promena

Kada napravite nove commit-ove na vašoj lokalnoj grani i želite da ih podelite sa timom ili ih sačuvate na udaljenom repozitorijumu (npr. GitHub), koristite komandu `git push`. Ona šalje vaše lokalne commit-ove na specificirani remote.

- **Svrha:** Postaviti lokalne commit-ove (i grane) na udaljeni repozitorijum.
- **Komanda:**

```
# Pošalji trenutnu granu na odgovarajuću remote granu (ako
git push

# Eksplicitno navođenje remote-a i grane
git push origin feature/user-login

# Slanje lokalne grane pod DRUGIM imenom na remote-u
```

`git push`: Deljenje vaših lokalnih promena 2

Podešavanje praćenja ("Upstream"): Kada prvi put šaljete novu lokalnu granu, Git ne zna sa kojom granom na remote-u treba da je poveže. Možete mu reći koristeći `-u` (ili `--set-upstream`) flag:

```
# Pošalji 'new-feature' na 'origin' i postavi praćenje
# tako da ubuduće 'git push/pull' na ovoj grani
# automatski koriste 'origin/new-feature'
git push -u origin new-feature
```

Posle ovoga, dovoljno je samo `git push` ili `git pull` dok ste na `new-feature` grani.

Brisanje remote grane:

```
# Starija sintaksa
git push origin :ime-remote-grane
# Novija, jasnija sintaksa
```

Force Push: Podsetnik na opasnosti i `--force-with-lease`

Ponekad, posebno nakon što ste uradili `rebase` na grani koja je *možda* već bila push-ovana (što generalno treba izbegavati!), standardni `git push` će biti odbijen (non-fast-forward). U takvim situacijama, možda ćete doći u iskušenje da koristite "force push". **Ovo zahteva izuzetan oprez.**

`git push --force` (ili `-f`): Bezuslovno gaženje!

Ova komanda govori udaljenom repozitorijumu: "Zanemari tvoju trenutnu istoriju za ovu granu i prihvati moju verziju kao jedinu ispravnu." **Izuzetno opasno** na deljenim granama! Ako je neko drugi push-ovao svoje promene od vašeg poslednjeg fetch-a, `--force` će **tiho obrisati** te promene sa remote-a. Taj rad može biti trajno izgubljen za tim.

Koristiti samo ako ste 1000% sigurni da niko drugi nije radio na toj grani i da vaša lokalna verzija zaista treba da pregazi remote.

Force Push: Podsetnik na opasnosti i `--force-with-lease` 2

`git push --force-with-lease`: Bezbednija alternativa!

Ova komanda je pametnija. Ona kaže: "Pregazi udaljenu granu mojom verzijom, **ALI SAMO AKO** je udaljena grana trenutno u stanju u kojem ja mislim da jeste (tj. u stanju od mog poslednjeg fetch-a).". Pre push-a, Git proverava da li SHA-1 heš udaljene grane odgovara onome što vaš lokalni `origin/<grana>` pamti.

Ako je neko drugi push-ovao u međuvremenu, SHA-1 se neće poklapati, i `push --force-with-lease` će biti **odbijen**, sprečavajući vas da slučajno obrišete tuđi rad.

Generalno, uvek koristite `--force-with-lease` umesto `--force` ako baš morate da pregazite remote istoriju (npr. nakon rebase-a *vaše lične* feature grane koju ste možda slučajno push-ovali ranije).

Podsetnik na pravilo: Najbolji način da izbegnete potrebu za force push-om jeste da **ne radite `rebase` na deljenim granama/commit-ovima.**

Tipičan Workflow Saradnje (Primer)

Kada radite u timu koristeći zajednički udaljeni repozitorijum (npr. `origin` na GitHub-u), uspostavlja se tipičan ciklus rada kako bi se kod efikasno delio i integrisao.

Početak rada na novom zadatku/feature-u

- **Ažurirajte glavnu granu:** Pre nego što počnete, osigurajte da imate najnoviju verziju glavne linije razvoja (npr. `main` ili `dev`).

```
git switch main
git pull origin main  # Ili git fetch + git merge origin/main
```

- **Kreirajte novu feature granu:** Odvojite novu granu iz ažurirane glavne grane (po potrebi iz druge grane).

```
git switch -c feature/user-password-reset main
```

Tipičan Workflow Saradnje (Primer) 2

Rad na feature grani

- Pravite izmene u kodu.
- Pravite commit-ove sa jasnim porukama

```
git add .
git commit -m "Implementiran deo X za resetovanje korisničke lozi
```

- Povremeno sačuvajte svoj rad na remote-u (ako želite backup ili ranu saradnju):

```
# Prvi put za ovu granu
git push -u origin feature/user-password-reset
# Kasnije
git push
```

Tipičan Workflow Saradnje (Primer) 3

Sinhronizacija sa glavnom granom (dok radite)

Da biste izbegli velike konflikte kasnije, povremeno integrišite najnovije promene iz glavne grane u vašu feature granu.

```
# (Dok ste na feature/user-password-reset)
# Opcija A: Rebase (ako grana nije deljena ili timski dogovor)
git fetch origin
git rebase origin/main
# Opcija B: Merge (bezbednije ako je grana deljena)
git fetch origin
git merge origin/main
```

Završetak rada i priprema za spajanje

- Uradite finalne commit-ove.
- (Opciono, ali preporučljivo): Sredite lokalnu istoriju commit-ova ako je potrebno (npr. `git rebase -i origin/main`).
- Ažurirajte granu poslednji put sa `origin/main` (koristeći `rebase` ili `merge`).

Tipičan Workflow Saradnje (Primer) 4

Predlaganje spajanja (npr. Pull Request na GitHub-u)

- Preko web interfejsa (GitHub, GitLab…), otvorite Pull Request (ili Merge Request) iz vaše `feature/user-password-reset` grane ka glavnoj grani (`main` ili `dev`).