Epizóda 1: Úvod do Gitu a GitHubu

🔷 Čo je Git a prečo ho používať?

***Git** je **distribuovaný verzovací systém**, ktorý slúži na správu zdrojového kódu. Umožňuje sledovať zmeny v súboroch, vracať sa k starším verziám a spolupracovať s viacerými ľuďmi na jednom projekte.*

Prečo používať Git?

*1. **História zmien** – Git uchováva všetky zmeny, takže sa môžeš kedykoľvek vrátiť k predchádzajúcej verzii kódu.*

*2. **Spolupráca v tíme** – Viacerí vývojári môžu pracovať na tom istom projekte bez toho, aby si prepisovali svoju prácu.*

*3. **Vetvenie (Branches)** – Každý môže pracovať na novej funkcionalite bez toho, aby narušil hlavný kód. Zmeny sa neskôr spoja cez **merge**.*

*4. **Bezpečnosť** – Ak niečo pokazíš, vieš sa kedykoľvek vrátiť k funkčnej verzii.*

*5. **Automatizácia a CI/CD** – Git je základom moderných vývojových workflow, napríklad **automatických testov** a **nasadzovania kódu (deployment)**.*

*6. **Bezplatné cloudové úložisko** – GitHub, GitLab alebo Bitbucket umožňujú ukladať a zdieľať repozitáre online.*

Git sa dnes používa takmer vo všetkých softvérových firmách a open-source projektoch, takže jeho znalosť je pre vývojárov nevyhnutná.

🔷 Rozdiel medzi Git a GitHub

Veľmi častá otázka, na ktorú sa dnes pozrieme!”

*Predstav si, že Git je ako **tvoj vlastný zápisník**, kde si zapisuješ všetky verzie svojho projektu. Môžeš sa kedykoľvek vrátiť k staršej verzii, porovnať zmeny alebo si odložiť rozrobenú prácu. Git funguje **lokálne**, priamo na tvojom počítači.*

*GitHub je naopak **online knižnica tvojich zápisníkov**. Je to cloudová služba, kde môžeš svoje projekty zdieľať, spolupracovať s inými vývojármi a zálohovať si kód.*

💡 ***Zjednodušene:***

*• **Git** = nástroj na verzovanie kódu (beží na tvojom PC).*

*• **GitHub** = online platforma na zdieľanie a spoluprácu s Git repozitármi.*

*A ešte jedna vec – GitHub **nie je jediný**. Existujú aj iné platformy, ako **GitLab, Bitbucket alebo Azure DevOps**, ktoré robia podobnú prácu. Ale GitHub je dnes najpopulárnejší, takže s ním budeme pracovať aj my.*

Takže, ak chceš len sledovať zmeny vo svojom kóde, Git ti úplne stačí. Ale ak chceš projekt zálohovať alebo na ňom pracovať s tímom, GitHub je nevyhnutný.

🔷 Inštalácia Gitu a základná konfigurácia (meno, e-mail)

Windows

1. Prejdi na stránku [git-scm.com](git-scm.com) a stiahni najnovšiu verziu pre Windows.

2. Spusti inštalátor a nechaj predvolené nastavenia.

*3. Po dokončení otvor **terminál (cmd) alebo Git Bash** a over, či Git funguje:*

```
git –version
```

Mac

*1. Otvor **Terminál** a zadaj:*

brew install git

Ak nemáš Homebrew, najskôr ho nainštaluj

git –version

Linux (Ubuntu/Debian)

1. Spusti príkaz:

sudo apt update && sudo apt install git

git –version

Po inštalácii je dôležité nastaviť meno a e-mail, ktoré sa budú zobrazovať pri každom commite.

Nastavenie mena:

git config --global user.name "TvojeMeno"

Nastavenie e-mailu:

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

Overenie nastavení:

git config --list

🔷 Vytvorenie prvého repozitára

Ako vytvoriť svoj prvý Git repozitár? Poďme na to!

Predstav si, že Git repozitár je ako špeciálny priečinok, kde Git sleduje všetky zmeny v tvojich súboroch. Keď niečo upravíš, Git to zaznamená a umožní ti kedykoľvek sa vrátiť späť. Takže, poďme si jeden taký repozitár vytvoriť!

Vytvoríme si priečinok.

mkdir moj-prvy-projekt

cd moj-prvy-projekt

*Teraz z tohto priečinka spravíme **Git repozitár**.*

git init

*Git nám vytvoril skrytý priečinok **.git**, kde bude sledovať všetky zmeny.*

Skontrolovať to môžeme príkazom:

ls -a # (na Linux/Mac)

dir /A # (na Windows)

🔷 Prvé commity: git init, git add, git commit

Teraz si v priečinku vytvoríme prvý súbor:

buď klasicky alebo podľa príkazu:

echo "Hello, Git!" > README.md

Git zatiaľ o tomto súbore nevie. Overíme to príkazom:

git status

*Uvidíme, že **README.md** je v červenej farbe – Git ho zatiaľ nesleduje.*

Pridáme ho do Gitu:

git add README.md

Ak chceme pridať všetky súbory, použijeme:

git add .

*Keď znova spustíme git status, uvidíme, že súbor je teraz **zelený** – Git ho pripravil na uloženie.*

Teraz všetko potvrdíme prvým commitom:

git commit -m "Prvý commit: pridanie README súboru"

Týmto sme oficiálne uložili prvú verziu nášho projektu!

A to je všetko! Práve sme vytvorili náš prvý Git repozitár, pridali súbor a uložili ho. V ďalšej časti si ukážeme, ako tento projekt nahrať na GitHub, aby sme ho mali v cloude a mohli na ňom pracovať z viacerých zariadení.

Epizóda 2: Práca s GitHubom – spojenie lokálneho projektu s cloudom

🔷 Prepojenie lokálneho repozitára s GitHubom (git remote add origin …)

Všetko nájdem na www.github.com

🔷 Nahranie kódu na GitHub (git push)

🔷 Stiahnutie zmien z GitHubu (git pull)

A je to! Lokálny projekt sme úspešne prepojili s GitHubom a nahrali ho online. Odteraz môžeme kód pravidelne aktualizovať jednoduchým git push. V ďalšej časti si ukážeme, ako stiahnuť nové zmeny z GitHubu späť do nášho počítača!

Ako stiahnuť zmeny z GitHubu pomocou git pull?

Predstav si, že pracuješ na svojom projekte a medzitým niekto iný – alebo aj ty sám z iného zariadenia – urobí zmeny v GitHub repozitári. Ako tieto zmeny dostať späť do svojho lokálneho počítača?

Na to slúži príkaz git pull! Poďme si to vysvetliť.

Najskôr sa uistíme, že sme v správnej vetve, do ktorej chceme stiahnuť zmeny.

V termináli zadaj:

git branch

*Ak vidíš * main (alebo * master), si na hlavnej vetve. Ak nie, prepni sa na ňu príkazom:*

git checkout main

alebo

git checkout master

*Teraz si načítame zmeny z **vzdialeného repozitára** (GitHubu) do nášho lokálneho počítača:*

git pull origin main

alebo ak používaš master vetvu:

git pull origin master

Čo sa deje?

- *Git sa spojí s GitHubom*
- *Skontroluje, či existujú nové zmeny*
- *Ak áno, stiahne ich a zlúči ich s tvojím kódom*

Riešenie konfliktov (ak nastanú)

Ak si ty aj niekto iný menil tie isté riadky v rovnakých súboroch, môže sa stať, že Git zobrazí konflikt.

V takom prípade uvidíš niečo ako:

```
CONFLICT (content): Merge conflict in subor.txt
```

👉*Otvor tento súbor a uvidíš v ňom časti označené <<<<<<<, ======= a >>>>>>>.*

```
<<<<<<< HEAD
Moja verzia kódu
=======
Verzia kódu z GitHubu
>>>>>>> origin/main
```

➡️*Uprav kód tak, aby zostala len správna verzia a uložené zmeny commitni:*

```
git add .
```

```
git commit -m "Riešenie konfliktu pri pull"
```

4️⃣*Overenie, že máme najnovšiu verziu*

Po úspešnom git pull môžeme skontrolovať, že máme najnovšie zmeny:

```
git log --oneline --graph --all
```

Ak vidíš najnovšie commity na vrchu, všetko je v poriadku! ✅

A je to! Teraz už vieš, ako jednoducho stiahnuť nové zmeny z GitHubu pomocou git pull. V ďalšej časti sa pozrieme na vetvenie v Gite – ako vytvoriť novú vetvu a prečo je to dôležité!

🔷 Klonovanie existujúceho repozitára (git clone)

Na klonovanie existujúceho repozitára pomocou Gitu použiješ nasledujúci príkaz:

```
git clone <url_repozitára>
```

Nahradíš <url_repozitára> URL adresou repozitára, ktorý chceš sklonovať. Napríklad:

```
git clone https://github.com/uzivatel/rep.git
```

Tento príkaz vytvorí kópiu repozitára do aktuálneho adresára. Vytvorí sa nový podadresár, ktorý bude obsahovať celý obsah repozitára.

Ak chceš klonovať repozitár do konkrétneho adresára, môžeš pridať názov adresára na konci:

```
git clone https://github.com/uzivatel/rep.git nazov_adresara
```

Týmto spôsobom sa klonovaný repozitár uloží do adresára nazov_adresara.

Epizóda 3: Práca s vetvami (Branches)

🔷 Prečo používať vetvy?

Vetvy (branches) v Gite sú kľúčovým nástrojom na správu rôznych verzií kódu a umožňujú efektívnejšiu spoluprácu na projektoch. Tu sú hlavné dôvody, prečo používať vetvy:

1. Izolácia práce

Vetvy umožňujú, aby si mohol pracovať na rôznych funkciách alebo opravách bez toho, aby si ovplyvnil hlavný kód v hlavnej vetve (zvyčajne main alebo master). Môžeš bezpečne vyvíjať, testovať a upravovať nový kód, bez toho aby to malo vplyv na stabilitu hlavnej verzie aplikácie.

2. Lepšia spolupráca v tíme

Viacero vývojárov môže pracovať na rôznych vetvách súčasne, pričom každá vetva môže mať svoj vlastný účel (napríklad na opravu chýb, pridávanie nových funkcií, refaktoring). Git umožňuje zlúčenie týchto vetiev (merging), čo zabezpečuje, že každý bude môcť prispieť, aniž by došlo k konfliktom v kóde.

3. Verzovanie a testovanie nových funkcií

Vetvy sa dajú využiť na experimentovanie alebo testovanie nových funkcií bez toho, aby si zasahoval do existujúceho kódu. Ak funkcia nebude fungovať podľa očakávaní, môžeš ju jednoducho zahodiť (vymazať vetvu) a návrat k stabilnej verzii bude bezproblémový.

4. Správa a organizácia projektov

Použitím vetiev si udržiavaš lepší prehľad o tom, na čom konkrétne pracuješ. Napríklad môžeš mať vetvu na opravu chýb (bugfix), vetvu na novú funkciu (feature-x), a vetvu na experimenty (experimental). Tento systém zjednodušuje orientáciu v kóde.

5. Zjednodušenie nasadzovania a integrácie

Keď je nová funkcia alebo oprava dokončená, môže sa zlúčiť s hlavnou vetvou. Tento proces je bezpečný, pretože vývoj na každej vetve prebieha izolovane. Môžeš tiež využívať ***pull requesty*** *alebo* ***merge requesty*** *na diskusiu a kontrolu kódu pred jeho slúčením.*

6. Lepšia správa verzií

Vetvy umožňujú jednoduchšiu správu rôznych verzií kódu. Ak si chceš uchovať verziu pred pridaním novej funkcie, jednoducho si vytvoríš vetvu a môžeš sa k nej vrátiť, keď to bude potrebné.

Používanie vetiev teda poskytuje flexibilitu, umožňuje bezpečné testovanie a udržiava projekty prehľadné a spravovateľné, najmä pri práci v tíme.

🔷 Základné operácie s vetvami:

Tu sú základné operácie s vetvami v Gite, ktoré sú veľmi užitočné pri práci na projekte:

1. Vytvorenie novej vetvy

Na vytvorenie novej vetvy použiješ príkaz:

```
git branch <nazov_vetvy>
```

Tento príkaz vytvorí novú vetvu, ale neprejdeš na ňu automaticky. Ak chceš hneď prepnúť na túto vetvu, použiješ:

```
git checkout <nazov_vetvy>
```

alebo kombinovaný príkaz na vytvorenie a okamžité prepnutie:

```
git checkout -b <nazov_vetvy>
```

2. Prepnúť sa na inú vetvu

Na prechod na existujúcu vetvu použiješ:

```
git checkout <nazov_vetvy>
```

Ak chceš prepnúť na vetvu a zároveň ju vytvoriť, použiješ vyššie uvedený príkaz s -b.

3. Zobrazenie zoznamu vetiev

Ak chceš zobraziť zoznam všetkých vetiev v repozitári, použiješ príkaz:

git branch

Aktuálna vetva bude označená hviezdičkou.

4. Zlúčenie vetvy (merge)

Po dokončení práce na svojej vetve môžeš zlúčiť zmeny späť do hlavnej vetvy (napr. main alebo master). Najprv prejdeš na vetvu, do ktorej chceš zmeny zlúčiť (napríklad main), a potom spustíš:

git merge <nazov_vetvy>

Tento príkaz zlúči zmeny z vetvy <nazov_vetvy> do aktuálnej vetvy. Ak existujú konflikty, Git ťa upozorní a budeš ich musieť vyriešiť.

5. Odstránenie vetvy

Ak už nepotrebuješ vetvu, môžeš ju odstrániť. Ak si na nej, najprv prejdite na inú vetvu a potom použite:

git branch -d <nazov_vetvy>

Tento príkaz odstráni vetvu, ale iba ak bola jej práca už zlúčená do inej vetvy. Ak ju chceš odstrániť aj v prípade, že nebola zlúčená, použiješ:

git branch -D <nazov_vetvy>

6. Presunutie zmien medzi vetvami (rebase)

Ak chceš upratať históriu alebo pridať zmeny z jednej vetvy na inú bez vytvárania merge commitov, môžeš použiť rebase:

git checkout <nazov_vetvy>
git rebase <nazov_vetvy_zdroja>

Tento príkaz aplikuje tvoje zmeny na vrch zmeneného histórie v inej vetve.

7. Kontrola, ktorá vetva má aké zmeny

Ak chceš vidieť, ktoré vetvy majú rôzne zmeny, môžeš použiť príkaz:

git diff <nazov_vetvy>..<nazov_vetvy>

Tieto operácie ti umožnia efektívne pracovať s vetvami v Gite a spravovať rôzne verzie kódu v tvojom projekte.

- ***Vytvorenie novej vetvy:*** *git branch názov-vetvy*
- ***Prepnutie medzi vetvami:*** *git checkout názov-vetvy (alebo git switch názov-vetvy)*
- ***Vytvorenie a prepnutie na novú vetvu:*** *git checkout -b názov-vetvy*
- ***Zobrazenie všetkých vetiev:*** *git branch*
- ***Zlúčenie vetvy do hlavnej vetvy:*** *git merge názov-vetvy*
- ***Vymazanie vetvy:*** *git branch -d názov-vetvy*

Epizóda 4: Riešenie konfliktov a návrat k predošlej verzii

🔷 Čo sú konflikty a ako ich riešiť?

Konflikty *v Gite nastávajú, keď sa v rôznych vetvách vykonajú zmeny na rovnakých riadkoch kódu alebo v rovnakých súboroch, a Git nedokáže automaticky rozhodnúť, ktoré zmeny by mali byť použité pri zlúčení (merge) alebo presune zmien (rebase). Tento problém sa zvyčajne objaví pri pokuse o zlúčenie dvoch vetiev, ktoré obsahujú vzájomne nezlučiteľné zmeny.*

1. Ako vznikajú konflikty?

Konflikty sa môžu vyskytnúť, keď:

• Dva vývojári upravujú rovnaký riadok v rovnakom súbore.

• Jeden vývojár odstráni alebo presunie súbor, ktorý bol upravený niekým iným.

• Pridáš nový súbor alebo vykonáš iné zmeny, ktoré nie sú kompatibilné so zmenami v inej vetve.

2. Ako Git indikuje konflikt?

*Pri pokuse o zlúčenie (merge) alebo rebase Git zistí konflikt a označí zasiahnuté súbory ako "**konfliktné"**. V súboroch, kde došlo ku konfliktu, Git vloží špeciálne značky na označenie konfliktu:*

• <<<<<<< HEAD: Tento riadok označuje kód v aktuálnej vetve (HEAD).

• =======: Tento riadok oddelí zmeny v oboch vetvách.

• >>>>>>> <nazov_vetvy>: Tento riadok označuje kód zo vzdialenej vetvy (t.j., tej, ktorú sa pokúšaš zlúčiť).

```
<<<<<<< HEAD
Tento text pochádza z mojej vetvy.
=======
Tento text pochádza z vetvy, ktorú chcem zlúčiť.
>>>>>>> feature-branch
```

3. Ako riešiť konflikty?

1. Identifikovanie konfliktov:

• Po pokuse o merge alebo rebase Git označí konfliktné súbory ako "unmerged".

• Môžeš ich zobraziť pomocou príkazu:

```
git status
```

Tento príkaz ti ukáže, ktoré súbory sú v konflikte.

2. Otvorenie a ručné riešenie konfliktu:

• Otvor každý konfliktný súbor v tvojom textovom editore.

• Prezri si označené oblasti (<<<<<<<, =======, >>>>>>>) a rozhodni sa, ktoré zmeny zachováš. Môžeš buď:

• Prijať zmeny z jednej vetvy.

• Kombinovať zmeny oboch vetiev.

• Urobiť úplne novú zmenu.

Po vybraní správnej verzie odstráň všetky značky konfliktu (<<<<<<<, =======, >>>>>>>).

3. Označenie konfliktu ako vyriešeného:

• Po vyriešení konfliktu a uložením súboru, označ tento súbor ako vyriešený:

```
git add <nazov_súboru>
```

4. Dokončenie merge/rebase:

• Po vyriešení všetkých konfliktov môžeš dokončiť merge:

```
git merge --continue
```

Alebo ak si robil rebase, môžeš pokračovať:

```
git rebase --continue
```

Ak by sa počas rebase objavili ďalšie konflikty, opakuj tento proces až do jeho dokončenia.

5. Alternatívy na riešenie konfliktov:

Použitie nástrojov na vizualizáciu konfliktov: Git ponúka rôzne nástroje na riešenie konfliktov (napr. git mergetool), ktoré ti môžu pomôcť vizuálne a pohodlnejšie riešiť konflikty.

Resetovanie alebo zrušenie zmien: Ak sa rozhodneš, že konflikty sú príliš zložité a nechceš ich riešiť, môžeš sa rozhodnúť pre obnovu pred merge pomocou:

```
git merge --abort
```

Alebo pre rebase:

```
git rebase --abort
```

6. Prevencia konfliktov:

• Pravidelne synchronizuj svoju vetvu s hlavnou vetvou, aby si znížil šancu na konflikty.

• Kombinuj menšie, častejšie zmeny namiesto veľkých a komplexných, ktoré môžu zvýšiť pravdepodobnosť konfliktov.

• Komunikácia v tíme: Uistite sa, že viacero vývojárov nepracuje na rovnakých častiach kódu naraz.

7. Riešenie konfliktov môže byť časovo náročné, ale je to nevyhnutná súčasť práce s Gitem pri spolupráci na projektoch. Dôležité je vždy byť opatrný a dôkladne skontrolovať, že po vyriešení konfliktov funguje aplikácia správne.

🔷 Návrat k predošlej verzii kódu:

Ak potrebuješ vrátiť k predchádzajúcej verzii kódu v Gite, máš niekoľko možností v závislosti od toho, akú verziu chceš obnoviť a čo všetko potrebuješ vrátiť (komit, súbor, celý projekt). Tu sú najbežnejšie spôsoby, ako sa vrátiť k predošlej verzii:

1. Vrátenie sa k predchádzajúcemu komitu

Ak chceš vrátiť celý projekt do stavu, v akom bol po predchádzajúcom komite (t. j. zrušiť posledné zmeny), použiješ príkaz git reset.

Nastavenie do predchádzajúceho komitu (soft reset)

Ak chceš vrátiť k predchádzajúcemu komitu, ale zachovať tvoje zmeny v pracovnom adresári (pridané alebo upravené súbory), použiješ:

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

Tento príkaz posunie HEAD o jeden komit späť, ale zmeny, ktoré boli vykonané medzi týmto komitom a aktuálnym stavom, zostanú v pracovnom adresári, pripravené na nové commity.

Resetovanie na predchádzajúci komit (hard reset)

Ak chceš úplne odstrániť posledné zmeny a vrátiť sa k predchádzajúcemu komitu (vrátane zmien v pracovnom adresári a indexe), použiješ príkaz:

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

Tento príkaz úplne zruší posledné zmeny a vráti projekt do stavu pred týmto komitom.

Vrátenie na konkrétny komit

Ak chceš vrátiť projekt na konkrétny komit, môžeš použiť príkaz s identifikátorom komitu (hash):

```
git reset --hard <commit_hash>
```

Tento príkaz nastaví HEAD na konkrétny komit, a vymaže všetky zmeny, ktoré boli vykonané po ňom.

2. Vrátenie jednotlivého súboru do predchádzajúcej verzie

Ak chceš vrátiť len konkrétny súbor do predchádzajúcej verzie (napríklad po nežiaducej zmene), môžeš použiť:

git checkout <commit_hash> -- <path_to_file>

Tento príkaz obnoví súbor na verziu z konkrétneho komitu. Ak chceš obnoviť súbor na verziu zo stavu pred posledným komitom, použiješ:

git checkout HEAD~1 -- <path_to_file>

3. Použitie git revert (vytvorenie nového komitu na zrušenie zmien)

Ak chceš "zrušiť" určité zmeny, ale nevymazať históriu (ako pri resetovaní), použiješ príkaz git revert. Tento príkaz vytvorí nový komit, ktorý zruší účinky konkrétneho komitu.

git revert <commit_hash>

Tento príkaz vytvorí nový komit, ktorý "vráti" predchádzajúce zmeny, ale zachová históriu neporušenú. Je to bezpečnejší spôsob, ako sa vrátiť k predchádzajúcej verzii, pretože históriu nezmazáva, len pridáva nový komit s opačnými zmenami.

4. Vrátenie zmeny v pracovnom adresári

Ak máš neuložené zmeny v pracovnom adresári (t. j. si ich ešte necommitol) a chceš sa vrátiť k predchádzajúcej verzii týchto súborov, použiješ:

git checkout -- <path_to_file>

Tento príkaz obnoví súbor na stav, v akom bol pri poslednom commite, a zruší všetky neuložené zmeny.

5. Vrátenie sa k predchádzajúcemu stavu pred merge/rebase

Ak si vykonal merge alebo rebase a chceš sa vrátiť späť pred túto operáciu, môžeš použiť príkaz git reflog, ktorý ti ukáže históriu HEAD a umožní ti nájsť bod, na ktorý sa chceš vrátiť.

git reflog

Tento príkaz vypíše históriu všetkých operácií s HEAD, a ty si môžeš vybrať konkrétny bod na vrátenie sa:

git reset --hard <commit_hash_from_reflog>

Zhrnutie

• *git reset --hard: Vráti sa na konkrétny komit a zruší všetky zmeny po ňom.*

• *git revert: Vytvorí nový komit, ktorý zruší zmeny konkrétneho komitu.*

• *git checkout: Vráti konkrétny súbor na predchádzajúcu verziu.*

• *git reflog: Zobrazuje históriu HEAD a umožňuje vrátiť sa k predchádzajúcemu stavu.*

• ***Zobrazenie histórie zmien:*** *git log --oneline*

• ***Návrat k predošlému commitu (bez straty zmien):*** *git checkout commit_ID*

• ***Resetovanie na konkrétny commit (s možnosťou ponechať alebo zahodiť zmeny):*** *git reset --soft commit_ID alebo git reset --hard commit_ID*

• ***Vytvorenie nového commitu na návrat späť':*** *git revert commit_ID*

Epizóda 5: Pokročilé techniky a užitočné tipy

🔹 ***Rebase vs. Merge*** *– kedy použiť čo?*

Pri práci s Gitem sú dve najbežnejšie operácie na zlúčenie zmien z rôznych vetiev ***merge*** *a* ***rebase****. Každá z týchto operácií má svoje výhody a nevýhody, a preto je dôležité pochopiť, kedy je vhodné použiť jednu alebo druhú. Dnes sa pozrieme na rozdiely medzi týmito dvoma prístupmi a na to, kedy je najlepšie použiť každý z nich.*

1. Merge

*Operácia **merge** je spôsob, ako spojiť dve vetvy do jednej. Git pri tejto operácii vytvorí **nový komit** na zlúčenie oboch vetiev. Merge zachováva históriu vetiev nezmenenú, čo znamená, že uvidíš všetky komity, ktoré sa vykonali na každej vetve.*

Výhody Mergu:

*• **Zachovanie histórie**: Merge neovplyvňuje existujúce komity, takže celý vývojový proces zostáva v histórii. Ak niekto v budúcnosti bude skúmať, čo sa v projekte stalo, bude jasné, ako a kedy sa rôzne vetvy spojili.*

*• **Bezpečné**: Je to jednoduchý príkaz na použitie a je veľmi bezpečný, keď pracuješ na verejných vetvách, ktoré zdieľajú rôzni vývojári.*

*• **Vhodné pre verejné vetvy**: Ak niekto pracuje na svojej vlastnej vetve a chce sa pridať späť do hlavnej vetvy, merge je ideálny, pretože nezmení históriu a jednoducho pridá nový komit.*

Kedy použiť Merge?

*• **Pri práci s verejnými vetvami** (napríklad main, develop), kde je dôležité zachovať históriu vývoja bez zmien.*

*• **Pri spájaní zmien od viacerých vývojárov**, pretože merge neprepíše žiadnu časť histórie, takže aj ostatní vývojári môžu ľahko pochopiť, čo sa stalo počas vývoja.*

*• **Ak nechceš meniť historické komity** a chceš mať jednoduchú operáciu, ktorá sa dá rýchlo vykonať.*

2. Rebase

*Rebase je trochu zložitejšia operácia, ktorá **presúva alebo aplikuje komity z jednej vetvy na druhú**. Na rozdiel od merge, rebase **mení históriu**. Predstav si to ako vytvorenie "nových" komitov, ktoré obsahujú rovnaké zmeny, ale sú aplikované na najnovší komit cieľovej vetvy.*

Výhody Rebase:

*• **Čistejšia história**: Rebase vytvára lineárnu históriu, čo znamená, že nebudú žiadne "merge commit" záznamy. V histórii nebudeš mať vetvy a zlúčenia, čo môže urobiť projekt prehľadnejším, najmä pri malých a rýchlych projektoch.*

*• **Jednoduchšie príspevky do hlavnej vetvy**: Ak pracuješ na svojej vlastnej funkcii v samostatnej vetve a pravidelne rebazuješ na main, vytvára to čistejší spôsob integrácie tvojich zmien bez zmätkov v histórii.*

*• **Použitie na osobných vetvách**: Rebase je veľmi užitočný na vetvách, kde pracuješ sám a chceš, aby histórie vyzerali lineárne a jednoduchšie na prehľadanie.*

Kedy použiť Rebase?

*• **Ak pracuješ na osobnej vetve a chceš mať čistú históriu** bez zbytočných merge commitov.*

*• **Pri upratovaní histórie pred pushovaním na vzdialený repozitár**. Ak chceš, aby všetky tvoje zmeny vyzerali, že boli vytvorené po sebe a nie v samostatných vetvách.*

*• **Keď chceš upravit históriu tak, aby vyzerala ako kontinuálny vývoj** – ideálne pre malé, osobné projekty alebo pred poslaním zmien do hlavného repozitára.*

3. Rozdiely medzi Merge a Rebase

*• **História**:*

*• **Merge**: Zachováva vetvy a ich históriu, pričom vytvorí nový "merge commit".*

*• **Rebase**: Presúva komity a vytvára čistejšiu, lineárnu históriu.*

*• **Zlúčenie zmien**:*

*• **Merge**: Vytvára nový commit, ktorý spája dve vetvy.*

*• **Rebase**: Vytvára nový commit na základe existujúcich zmien, ale na novom mieste v histórii.*

• ***Riziko konfliktov**:*

• ***Merge**: Konflikty sa riešia pri merge a vytvára sa nový commit.*

• ***Rebase**: Konflikty sa riešia počas rebase, čo môže byť komplikovanejšie.*

4. Kedy sa neodporúča používať Rebase?

• ***Na verejných vetvách**, kde pracuje viac vývojárov. Použitím rebase sa mení história a môže to spôsobiť problémy pre ostatných vývojárov, ktorí sa pokúšajú získať aktuálnu verziu.*

• ***Ak nie si si istý**, či rebase vykonáš správne, pretože môžeš zmeniť históriu spôsobom, ktorý bude ťažké opraviť.*

Zhrnutie:

• ***Používaj Merge**, keď chceš zachovať históriu, pracuješ s verejnými vetvami alebo spolupracuješ s inými vývojármi na rovnakých vetvách.*

• ***Používaj Rebase**, keď chceš čistejšiu, lineárnu históriu a robíš to na svojej osobnej vetve, kde meníš históriu, aby vyzerala ako kontinuálny vývoj.*

Toto je skvelý nástroj pre každého vývojára, ale nezabúdaj na to, že správna voľba medzi merge a rebase závisí od tvojich potrieb a situácie v projekte.

🔹 ***Stashing** – ako si odložiť rozrobené zmeny? (git stash)*

*Pri práci s Gitem sa môže stať, že potrebuješ na chvíľu opustiť svoje rozrobené zmeny a prepnúť sa na niečo iné, ale nechceš ich ešte commitovať. Tu prichádza na scénu **git stash** – nástroj, ktorý ti umožní dočasne uložiť tvoje rozrobené zmeny bez toho, aby si ich commitoval alebo zmazal.*

Čo je to git stash?

***git stash** je príkaz, ktorý **dočasne uloží tvoje necommitované zmeny (zmeny v pracovnom adresári a v indexe)** a **vráti projekt do čistého stavu**. Potom môžeš pokračovať v práci na inej úlohe, bez toho, aby si musel commitovať svoje nezávislé zmeny.*

Kedy použiť git stash?

• *Ak máš neuložené zmeny, ktoré ešte nechceš commitovať, ale potrebuješ prepnúť vetvy alebo urobiť niečo iné.*

• *Ak chceš na chvíľu "zabudnúť" na svoje aktuálne zmeny, ale nechceš ich stratiť.*

• *Ak pracuješ na dlhšom feature, ale musíš prepnúť vetvu, aby si napríklad opravil bug.*

Ako použiť git stash?

*1. **Uloženie zmien do stashu:***

Na uloženie tvojich zmien do stashu použiješ príkaz:

```
git stash
```

Tento príkaz uloží tvoje zmeny a vráti pracovný adresár do posledného commitnutého stavu.

*Ak chceš uložiť aj zmeny v **staging area (indexe)**, použiješ:*

```
git stash -k  # Alebo git stash --keep-index
```

*2. **Zobrazenie uložených stashingov:***

Ak chceš vidieť zoznam všetkých uložených stashingov, použiješ:

```
git stash list
```

Tento príkaz ti ukáže všetky stashované položky s ich popisom a indexom.

*3. **Obnovenie zmien zo stashu:***

Keď si chceš vrátiť svoje uložené zmeny, použiješ:

git stash apply

Tento príkaz vráti poslednú uloženú stash položku do tvojho pracovného adresára. Ak chceš aplikovať konkrétnu stash, môžeš zadať index:

git stash apply stash@{2}

4. ***Obnovenie zmien a odstránenie stashu:***

Ak chceš obnoviť zmeny a zároveň odstrániť stash z histórie, použiješ:

git stash pop

Tento príkaz obnoví tvoje zmeny a zároveň odstráni stash, ktorý si použil.

5. ***Odstránenie stash položky:***

Ak už nepotrebuješ konkrétnu stash, môžeš ju vymazať:

git stash drop stash@{1}

6. ***Vymazanie všetkých stash položiek:***

Ak chceš vymazať všetky uložené stashingy, použiješ:

git stash clear

Príklady použitia:

• Odloženie zmien a prechod na inú vetvu:

Predstav si, že pracuješ na nejakých úpravách, ale musíš rýchlo prepnúť na inú vetvu, aby si vyriešil nejaký urgentný problém. Môžeš uložiť svoje zmeny do stashu a neskôr sa k nim vrátiť:

git stash # Uloží aktuálne zmeny

git checkout bug-fix # Prejdeš na vetvu s opravou bugu

• Obnovenie stashovaných zmien:

Keď už opravíš bug a chceš sa vrátiť k svojim predchádzajúcim zmenám:

git checkout feature-branch # Prejdeš späť na svoju pôvodnú vetvu

git stash apply # Obnoví uložené zmeny

Zhrnutie:

git stash *je veľmi užitočný nástroj na dočasné uloženie zmien, ktoré nechceš commitovať. Pomôže ti rýchlo prepnúť vetvy alebo vykonať iné operácie bez toho, aby si stratil rozpracované zmeny. Pamätaj však, že stash je len dočasné riešenie a je dobré pravidelne commitovať svoje zmeny, aby si udržal čistú históriu v repozitári.*

🔹 ***Tagovanie verzií*** *(git tag v1.0)*

Tagovanie verzií v Gite je veľmi užitočné na označenie konkrétnych bodov v histórii repozitára, ktoré predstavujú významné verzie projektu, ako napríklad vydanie novej verzie alebo stabilného komitu. Používanie tagov ti pomáha udržať prehľad o tom, ktorá verzia kódu bola nasadená alebo ktorá verzia obsahuje špecifické funkcie.

Čo je to git tag?

Git tag je spôsob, ako priradiť štítok (tag) k určitému commit-u. Tento tag sa môže používať na označenie verzií alebo dôležitých okamihov v projekte. Tagy nie sú ako vetvy, pretože nie sú určené na ďalšie zmeny, ale slúžia ako statické označenie pre špecifické verzie.

Kedy používať git tag?

• Keď vydávaš novú verziu aplikácie alebo knižnice.

• Na označenie stabilných verzií alebo dôležitých bodov v histórii projektu.

• Ak chceš mať ľahký prístup k konkrétnym verziám kódu.

Ako používať git tag?

1. Vytvorenie tagu

Ak chceš vytvoriť nový tag pre konkrétny commit, jednoducho použiješ príkaz:

```
git tag v1.0
```

Týmto príkazom vytvoríš tag s názvom v1.0 na aktuálnom commit-e. Tento tag bude priradený k poslednému commit-u.

Ak chceš vytvoriť tag na konkrétny commit, môžeš zadať aj hash commitu:

```
git tag v1.0 abc1234
```

Kde abc1234 je hash commit-u, ku ktorému chceš priradiť tag.

2. Vytvorenie anotovaného tagu

Anotované tagy obsahujú viac informácií (napríklad autora, dátum, správu a môžu byť podpísané). Anotovaný tag vytvoríš príkazom:

```
git tag -a v1.0 -m "Verzia 1.0: Prvý stabilný release"
```

Príkaz -a znamená, že ide o anotovaný tag, a -m umožňuje pridať správu k tagu.

3. Zobrazenie tagov

Ak chceš zobraziť všetky tagy v repozitári, použiješ príkaz:

```
git tag
```

Tento príkaz zobrazí zoznam všetkých tagov.

Ak chceš zobraziť tagy, ktoré zodpovedajú určitému vzoru (napríklad verzie 1.x), môžeš použiť:

```
git tag -l "v1.*"
```

4. Pushing tagu na vzdialený repozitár

Tagy sa neodosielajú automaticky pri pushovaní zmien na vzdialený repozitár, musia sa explicitne odoslať. Pre odoslanie konkrétneho tagu použiješ:

```
git push origin v1.0
```

Tento príkaz pošle tag v1.0 na vzdialený repozitár.

Ak chceš poslať všetky tagy, ktoré ešte neboli odoslané, použiješ:

```
git push --tags
```

5. Prechádzanie medzi tagmi

Ak chceš prejsť na stav kódu označený konkrétnym tagom, použiješ:

```
git checkout v1.0
```

Týmto príkazom sa prepneš na verziu označenú tagom v1.0.

6. Odstránenie tagu

Ak chceš odstrániť tag, použiješ príkaz:

```
git tag -d v1.0
```

Tento príkaz odstráni tag lokálne. Ak chceš odstrániť tag aj zo vzdialeného repozitára, použiješ:

```
git push --delete origin v1.0
```

Príklady použitia:

1. Vytvorenie a pushovanie tagu po vydaní novej verzie:

Po dokončení nového vydania aplikácie chceš označiť verziu ako v1.0:

```
git tag v1.0
```

```
git push origin v1.0
```

2. Vytvorenie anotovaného tagu pre stabilnú verziu:

Ak vydávaš stabilnú verziu s popisom:

```
git tag -a v1.0 -m "Stabilná verzia 1.0"
```

```
git push origin v1.0
```

3. Zobrazenie a prechod na predchádzajúci tag:

Ak chceš zobraziť všetky tagy a potom prejsť na predchádzajúcu verziu:

```
git tag
```

```
git checkout v0.9
```

Zhrnutie:

Tagovanie verzií je skvelý spôsob, ako udržať prehľad o dôležitých bodoch v histórii tvojho projektu. Použitím tagov môžeš označiť verzie, ktoré sú stabilné alebo pripravené na nasadenie, čím zjednodušíš prácu s históriou kódu a ľahko sa vrátiš k predchádzajúcim verziám. Tagy sú veľmi užitočné najmä pri správe verzií a pri vydávaní nových verzií aplikácie.

🔹 ***Git aliasy** – skrátenie príkazov*

***Git aliasy** sú skvelý spôsob, ako si skrátiť a zjednodušiť najčastejšie používané príkazy v Gite. Pomocou aliasov môžeš ušetriť čas a zefektívniť svoju prácu, najmä ak pravidelne používaš dlhé alebo zložité príkazy.*

Čo sú git aliasy?

***Git aliasy** sú vlastné skrátené príkazy, ktoré môžeš nastaviť, aby si nemusel zakaždým písať dlhé príkazy. Môžeš si vytvoriť alias pre akýkoľvek príkaz Git-u, čím zjednodušíš prácu a zvýšiš svoju produktivitu.*

Kedy používať git aliasy?

• Ak pravidelne používaš dlhé príkazy, ktoré sa ti neoplatí každodenne písať.

• Ak chceš mať príkazy prispôsobené svojim potrebám.

• Ak chceš zrýchliť prácu v príkazovom riadku.

Ako nastaviť git aliasy?

1. Vytvorenie aliasu

*Alias môžeš nastaviť jednoducho pomocou príkazu **git config**. Tu je základný syntax pre vytvorenie aliasu:*

```
git config --global alias.[alias_name] "[original_command]"
```

Týmto príkazom si vytvoríš nový alias, ktorý bude k dispozícii vo všetkých tvojich repozitároch. Ak chceš alias nastaviť len pre konkrétny repozitár, vynecháš --global.

***Príklad 1**: Skrátenie príkazu pre **git status**:*

git config --global alias.st status

Teraz namiesto git status môžeš použiť príkaz git st.

***Príklad 2**: Skrátenie príkazu pre **git commit -m**:*

git config --global alias.cm "commit -m"

Tento alias umožní použiť príkaz git cm "tvoja správa", čo je rýchlejšie než písanie celého príkazu.

2. Vytvorenie komplexného aliasu

*Môžeš vytvoriť alias aj pre zložitejšie príkazy. Napríklad, ak chceš vytvoriť alias pre **git log –oneline –graph –decorate –all** (príkaz, ktorý zobrazuje graf histórie commitov), môžeš to urobiť takto:*

git config --global alias.lg "log --oneline --graph --decorate --all"

Potom môžeš použiť skrátený príkaz git lg namiesto celého príkazu.

3. Alias pre príkazy s viacerými parametrami

Ak chceš vytvoriť alias, ktorý obsahuje parametre, môžeš to spraviť bez problémov:

git config --global alias.last "log -n 1 HEAD"

Tento alias spustí príkaz na zobrazenie posledného commitu: git last.

4. Alias pre príkazy, ktoré obsahujú viacero častí (shell príkazy)

Ak chceš vytvoriť alias, ktorý spustí viacero príkazov alebo použije shell, môžeš to nastaviť aj takto:

git config --global alias.cleanup "!git gc && git prune"

*Tento alias spustí príkazy **git gc** (garbage collection) a **git prune** (odstránenie nezreferencovaných objektov) v jednom kroku pomocou git cleanup.*

5. Zobrazenie nastavených aliasov

Ak chceš zobraziť všetky nastavené aliasy, môžeš použiť príkaz:

git config --get-regexp alias

Príklady užitočných git aliasov:

*1. **git co** - pre skracovanie príkazu git checkout:*

git config --global alias.co checkout

*2. **git br** - pre zobrazenie zoznamu vetiev:*

git config --global alias.br branch

*3. **git cm** - pre commit s správou:*

git config --global alias.cm "commit -m"

*4. **git lg** - pre zobrazenie histórie v grafe:*

git config --global alias.lg "log --oneline --graph --decorate --all"

*5. **git st** - pre skrátenie príkazu git status:*

git config --global alias.st status

*6. **git unstage** - pre odstránenie súboru zo staging area:*

```
git config --global alias.unstage "reset HEAD"
```

*7. **git amend** - pre rýchle upravenie posledného commit-u:*

```
git config --global alias.amend "commit --amend"
```

Ako odstrániť git alias?

Ak sa rozhodneš, že už nechceš používať určitý alias, môžeš ho odstrániť pomocou príkazu:

```
git config --global --unset alias.[alias_name]
```

Príklad:

```
git config --global --unset alias.co
```

Zhrnutie:

***Git aliasy** sú efektívny spôsob, ako si skrátiť a prispôsobiť príkazy v Gite. Pomocou aliasov môžeš šetriť čas pri práci a prispôsobiť Git svojim potrebám. Môžeš vytvárať aliasy pre jednorazové príkazy alebo pre časté operácie, čím zlepšíš svoju produktivitu a zjednodušíš prácu s Gitom.*