

# Git - Uživatelská konfigurace



## Povolení dlouhých cest ve Windows

```
git config --system core.longpaths true
```

povolí v Git podporu dlouhých cest na Windows, což často řeší chybu „Filename too long“.

### ⚠ Pozor:

- Tento příkaz se musí spustit s administrátorskými právy, protože mění systémovou konfiguraci Gitu.
- Musí mít ve Windows povolenou podporu dlouhých cest. (Pokud to není povolené, Git to nezvládne.)

Pokud ještě nemáte povolené dlouhé cesty v systému, lze to udělat takto:



1. Spustí `regedit`
2. Najdi klíč: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem`
3. Najdi nebo vytvoř DWORD hodnotu `LongPathsEnabled` a nastav ji na `1`.
4. Restartuj počítač.



## Nastavení Meld jako diff/merge tool



**Meld** je vizuální nástroj pro porovnávání a slučování souborů.

Umožňuje přehledné zobrazení rozdílů a snadné řešení konfliktů.

- ▶  [Windows – Kompletní postup](#)
- ▶  [Linux – Kompletní postup](#)



## Použití v praxi

- ▶  [Porovnání změn](#)
- ▶  [Řešení konfliktů při slučování](#)



# Git – Práce s úložištěm



Praktické rady pro vytvoření a použití Git úložiště na lokálním i online prostředí.

---



## Vytvoření úložiště

►  [Kompletní postup](#)

---



## Klonování úložiště

►  [Použití v pracovním prostředí](#)



# Git – Submoduly: Přehled & použití



Praktické rady pro správu externích repozitářů pomocí submodulů v Git.

---



## Co jsou submoduly?

- Umožňují vložit jeden Git repozitář do jiného jako podadresář.
  - Zachovávají nezávislost obou repozitářů.
  - Vhodné pro správu závislostí, sdílený kód nebo více projektů najednou.
- 






## Struktura projektu se submodulem

- ▶  Ukázková struktura
- 



## Základní příkazy

- ▶  Přidání submodulu
  - ▶  Klonování projektu se submodule
  - ▶  Aktualizace submodulů
- 






## Praktický příklad v Unity projektu

- ▶  Struktura & přidání submodulů
- 




## Tipy pro práci se submodule

- ▶  Přepínání verzí submodulu
  - ▶  Úpravy v submodule
  - ▶  Odstranění submodulu
- 



## Řešení častých problémů

- ▶  Submodul v "detached HEAD" stavu
- ▶  Submodul ukazuje změny i když žádné nemáte

- ▶  Změna URL submodulu
  - ▶  Kontrola stavu submodulů
- 

## **Výhody & nevýhody submodulů**

- ▶  Výhody
- ▶  Nevýhody



# Git Flow – Strategie větvení & workflow



Praktické rady pro efektivní správu větví v týmu pomocí Git Flow.

---





## Co je Git Flow?

- **Git Flow** je osvědčená strategie pro řízení verzí a vývoj v týmech.
  - Umožňuje jasně oddělit vývoj, přípravu vydání a opravy chyb.
- 






## Hlavní větve

- ▶  Základní větve
  - ▶  Pomocné větve
- 



## Typické workflow

- ▶  Vývoj nové funkce
  - ▶  Příprava vydání
  - ▶  Oprava chyby v produkci
- 



## Pravidla pro práci s Git Flow

- ▶  Doporučené postupy
- 



## Vizualizace workflow

- ▶  Schéma větvení



# Praktické použití gitu



## Proč někdy přepisujeme vzdálenou větev?

- Po `rebase` nebo `commit --amend` se mění historie větve.
- Při push může vzniknout chyba `non-fast-forward` – remote větev má nové commity, které nejsou v lokální větvi.
- Chceme zachovat vlastní změny, ale **nechceme přijít o cizí práci**.



## Jak Git chrání historii?

- **Non-fast-forward** push je zablokován, aby se nechtěně nepřepsaly cizí commity.
- Git vyžaduje explicitní potvrzení, že víš, co děláš.



## Doporučený workflow krok za krokem

### 1 Aktualizuj si remote:

```
git fetch origin
```



Získáš aktuální stav vzdálené větve.

### 2 Proved' změny (např. `rebase`, `commit --amend`)

### 3 Bezpečně pushni změny:

```
git push origin main --force-with-lease
```



Přepíše vzdálenou větev **jen pokud se nezměnila od tvého posledního fetch/pullu**.

## ! Co dělat při chybě?

- Push s `--force-with-lease` selže, pokud někdo mezitím pushnul nové změny.
- Stáhni je (`git fetch`), vyřeš konflikty a workflow opakuj.

## Rizika & doporučení

- 💣 `git push --force` přepíše remote bez kontroly – použij **jen pokud jsi jediný na větvi!**
  - 🗨️ Vždy informuj tým, pokud musíš přepisovat historii.
- 

## Slovníček pojmů

- ▶️ **Fast-forward:** Push bez konfliktu, remote větev je přímo navazující.
- 🚫 **Non-fast-forward:** Remote větev má nové commity, které nejsou v tvé větvi.
- 💣 **Force push:** Přepíše remote větev bez kontroly.
- 🛡️ **Force-with-lease:** Přepíše remote větev jen pokud se nezměnila od tvého posledního fetch/pullu.



# Git – Vytvoření & push nové větve na remote

🚀 Praktické rady pro založení a umístění nové větve (`develop`) na Git server (např. GitHub, GitLab).

---



## Vytvoření nové větve

- ▶ 🛠️ [Krok 1: Založení větve](#)
- 



## Nastavení vzdáleného repozitáře

- ▶ 🔗 [Krok 2: Ověření remote](#)
- 



## Push větve na server

- ▶ 📦 [Krok 3: Push větve na remote](#)
- 



## Ověření online

- ▶ 🔍 [Krok 4: Kontrola na webu](#)





# Git – Smazání vzdálené větve



Praktické rady pro bezpečné odstranění větve z Git serveru (např. GitHub, GitLab).



## Upozornění



### Warning

Smazání vzdálené větve je **nevratná operace**.

Ujisti se, že větev už nepotřebuješ a všechny důležité změny jsou začleněny jinde.



## Postup krok za krokem

- ▶ 🔍 Krok 1: Zobrazení všech větví
- ▶ 🗑️ Krok 2: Smazání vzdálené větve
- ▶ 🧹 Krok 3: Vyčištění lokálních referencí



# Git – Pull Request (PR)



Praktické rady, jak funguje **Pull Request**, kdy a proč ho použít.

---







## Co je Pull Request?

- **Pull Request (PR)** je žádost o začlenění změn z jedné větve do jiné (typicky z `feature` do `main` nebo `develop`).
  - Umožňuje týmovou kontrolu, diskusi, testování a schválení změn před sloučením.
- 



## Typický workflow PR

- ▶  Krok 1: Vytvoření nové větve
  - ▶  Krok 2: Vytvoření PR na serveru
  - ▶  Krok 3: Code review & testy
  - ▶  Krok 4: Schválení & merge PR
- 



## Výhody Pull Requestu

- ▶  Proč používat PR?



# Git – Aktualizace `.gitignore` & odstranění mezipaměti



Praktické rady, jak správně aktualizovat `.gitignore` a odstranit již sledované soubory z Git mezipaměti.

---






## Proč aktualizovat `.gitignore`?

- `.gitignore` určuje, které soubory Git nemá sledovat.
  - Po změně je nutné odstranit již sledované soubory z mezipaměti, aby se ignorovaly.
- 






## Postup krok za krokem

- ▶  Krok 1: Odstranění mezipaměti sledovaných souborů
- ▶  Krok 2: Přidání změn do stage
- ▶  Krok 3: Commit změn




# Git – Přesun commitů do nové nebo existující větve

 Praktické rady, jak přesunout poslední commity ze jedné větve do nové nebo existující větve.

## Přesun commitů do nové větve

- ▶  Krok 1: Vytvoření nové větve z aktuální
- ▶  Krok 2: Odstranění commitů ze zdrojové větve
- ▶  Krok 3: Přepnutí do nové větve

## Přesun commitů do existující větve

- ▶  Krok 1: Merge commitů do cílové větve
- ▶  Krok 2: Odstranění commitů ze zdrojové větve
- ▶  Krok 3: Přepnutí do cílové větve

### Note

Více informací najdeš v [diskuzi na Stack Overflow](#).



# Git – **fixup!** & **squash!** commity

🚀 Praktické rady, jak efektivně opravovat a slučovat commity pomocí **fixup!** a **squash!** v Gitu.

---



## Co znamená **fixup!** a **squash!**?

- **fixup!** – vytvoří commit, který opravuje předchozí commit bez změny jeho zprávy.
  - **squash!** – vytvoří commit, který sloučí zprávu s původním commitem.
- 



## Postup krok za krokem

- ▶ 🔧 Krok 1: Vytvoření opravného commitu
- ▶ ↺ Krok 2: Rebase s automatickým sloučením
- ▶ ✎ Krok 3: Úprava v editoru

# Git – Nahrazení vzdálené větve z lokální větve







 Praktický návod, jak kompletně nahradit historii vzdálené větve pomocí nové lokální větve.

---

## Kdy použít tento postup?

- Chceš začít s čistou historií commitů (např. po refaktoringu).
  - Potřebuješ odstranit veškerou předchozí historii z hlavní větve (`main` / `master`).
  - Vhodné pro projekty, kde je nutné kompletní "reset" repozitáře.
- 

## Postup krok za krokem

- ▶  Krok 1: Vytvoření nové větve bez historie
- ▶  Krok 2: Přidání všech souborů
- ▶  Krok 3: První commit
- ▶  Krok 4: Smazání původní hlavní větve
- ▶  Krok 5: Přejmenování nové větve na hlavní
- ▶  Krok 6: Force push do vzdáleného repozitáře

Zde je moderní návod v 3. osobě pro soubor `vcs/git/history/assumeUnchanged.md`:



# Git – Lokální ignorování změn (**assume-unchanged**)

 Praktické rady, jak lze **lokálně ignorovat změny** ve složce nebo souborech, aniž by se to projevilo na vzdáleném repozitáři.





## Kdy použít?

- Pokud je potřeba **fyzicky smazat** nebo upravit soubory/složku pouze lokálně, ale **neovlivnit remote** při push.
- Když má Git **ignorovat změny** v konkrétních souborech/složkách pouze na daném počítači.



## Postup krok za krokem

- ▶  Ignorování změn ve složce/souborech
- ▶  Vrácení zpět (opětovné sledování změn)



## Upozornění na rizika

- **Změny jsou pouze lokální** – ostatní členové týmu je nevidí.
- Pokud se commitne smazání nebo úprava souborů, push už remote ovlivní.
- Vhodné pro dočasné úpravy, testování, nebo když je potřeba něco skrýt před Gitem.



## Slovníček pojmů

- **assume-unchanged**: Git ignoruje změny v souboru/složce pouze na daném počítači.
- **no-assume-unchanged**: Git opět začne změny sledovat.



## Shrnutí

- Lokální ignorování změn je **bezpečné** pro push, dokud se necommitne smazání nebo úprava.
- Remote repozitář zůstane **beze změny**.

---

**i Tip**

Pro trvalé ignorování se používá `.gitignore`, pro dočasné lokální ignorování slouží `assume-unchanged`.