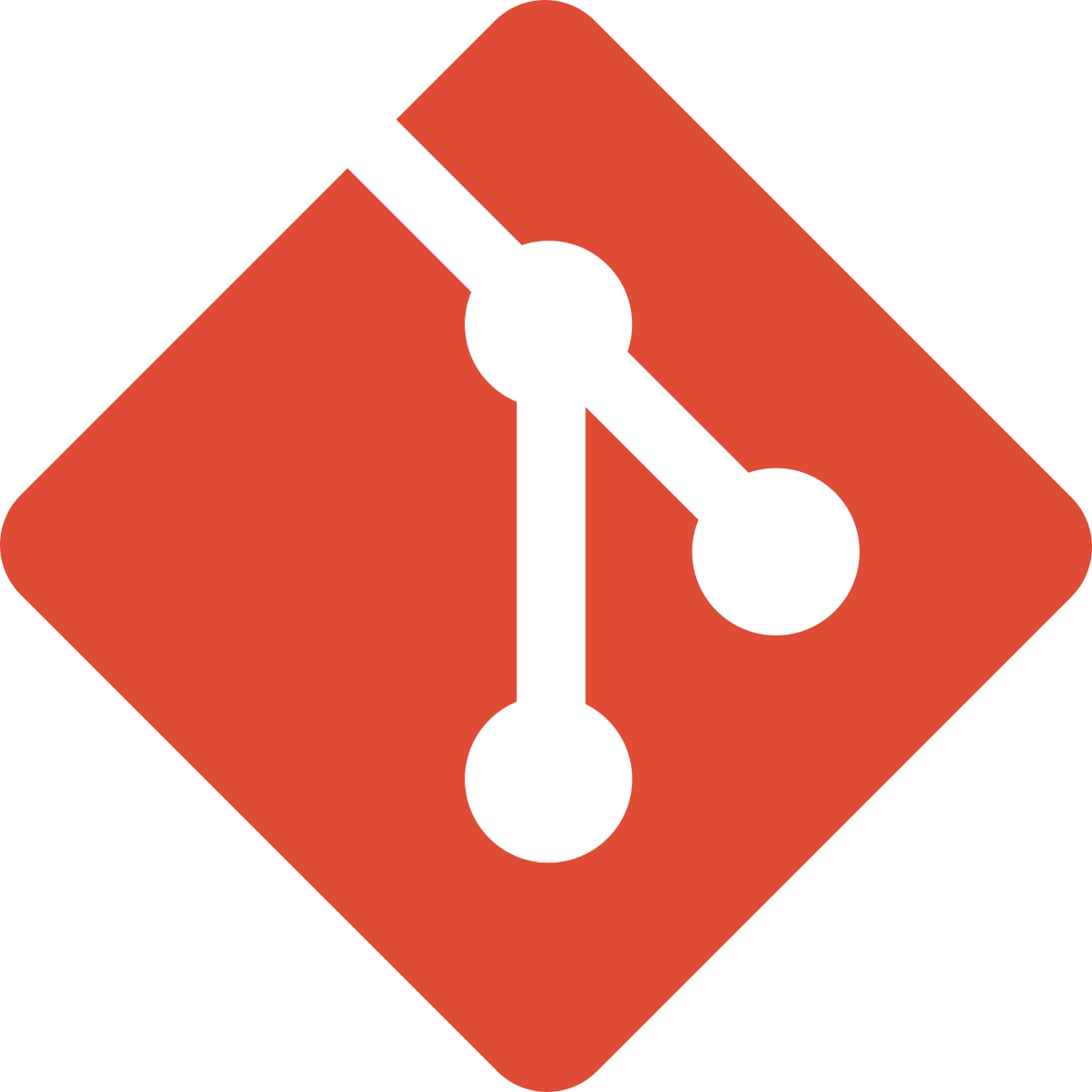
Git kisokos

NAGYOKNAK IS

Gyulai Balázs



Tartalom

[Mi a franc ez? 0](#_Toc179737967)

[0.1 Mi az a verziókezelés? 0](#_Toc179737968)

[0.2 Manuális (falusi) verziókezelés 0](#_Toc179737969)

[0.3 Centralized Version Control Systems 1](#_Toc179737970)

[0.4 Distributed Version Control Systems 1](#_Toc179737971)

[Git 0](#_Toc179737972)

[1. Letöltés és telepítés 0](#_Toc179737973)

[2. Projekt létrehozása 5](#_Toc179737974)

[2.1. git init 5](#_Toc179737975)

[3. Követem vagy nem követem? 6](#_Toc179737976)

[3.1. git status 6](#_Toc179737977)

[3.2. git add 7](#_Toc179737978)

[3.3. git rm –cached 8](#_Toc179737979)

[4. Commitok 9](#_Toc179737980)

[Vizuálisan 9](#_Toc179737981)

[5. Előzmények 11](#_Toc179737982)

[5.1. git log 11](#_Toc179737983)

[6. Branchek 12](#_Toc179737984)

[Vizuálisan 12](#_Toc179737985)

[7. Branchek és a mergelés 15](#_Toc179737986)

[7.1. git merge 15](#_Toc179737987)

[7.2. git rebase 17](#_Toc179737988)

[8. Mozgás Gitben 19](#_Toc179737989)

[8.1. HEAD 19](#_Toc179737990)

[8.2. Relatív Refs 21](#_Toc179737991)

[9. Branch forcing 24](#_Toc179737992)

[Vizuálisan 24](#_Toc179737993)

[10. Változtatások visszaállítása 26](#_Toc179737994)

[10.1. git reset 26](#_Toc179737995)

[10.2. git revert 28](#_Toc179737996)

[11. Mozgasd a munkád 30](#_Toc179737997)

[11.1. git cherry-pick 30](#_Toc179737998)

[11.2. git interactive rebase 31](#_Toc179737999)

[12. Távoli repository 34](#_Toc179738000)

[12.1. git clone 34](#_Toc179738001)

[12.2. Távoli branchek 35](#_Toc179738002)

[12.3. git fetch 36](#_Toc179738003)

[12.4. git pull 38](#_Toc179738004)

[12.5. git remote 39](#_Toc179738005)

[12.6. git config 39](#_Toc179738006)

[12.6. git push 40](#_Toc179738007)

[12.7. Verzió eltérés 40](#_Toc179738008)

[13. Távoli repo frissítés eldobás 44](#_Toc179738009)

[13.1. Miért dobta el? 44](#_Toc179738010)

[13.2. Megoldás 44](#_Toc179738011)

[Források és ajánló 0](#_Toc179738012)

[Ábrajegyzék 0](#_Toc179738013)

# Mi is ez valójában?

## 0.1 Mi az a verziókezelés?

A verzió kezelés egy rendszer, ami időről időre rögzíti fájlok változtatásait, ezáltal visszalehet állítani a fájlt egy korábbi verzióra. Verzió kezelő rendszert (Version Control System, VSC) nagyon bölcs dolog használni, mivel megkönnyíti a mindennapokat és gyorsítja egy projekt fejlesztését is. Azért is jó egy ilyen rendszert használni, mivel, ha elveszik valami egy fájlon és nincs kedved, türelmed újrakezdeni, akkor nagyon egyszerűen vissza lehet állítani.

## 0.2 Manuális (falusi) verziókezelés

Rengeteg embernek az a rendszere, hogy állományokat egy pl. időbélyeggel elnevezett mappába tesznek, mivel ez nagyon egyszerűen hangzik, viszont annál nagyobb az esély a hibázásra.

Nagyon könnyű elfelejteni, melyik mappában van az ember és ezáltal rossz helyre történik, a fájlok másolása.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

1. ábra – Local Version Control Systems

## 0.3 Centralized Version Control Systems

A központosított verzió kezelő rendszert (Centralized Version Control Systems, CVCS) még nagyon régen okos fejlesztők kidolgozták, ami nem más, mint egy távoli megosztott mappa, amit több másik munkatárs is tudott szerkeszteni. Sok-sok ideig ez volt a megszokott mód a verziókezelésre.

A képen szöveg, Betűtípus, sor, tervezés látható

Automatikusan generált leírás

2. ábra – Centralized Version Control Systems

## 0.4 Distributed Version Control Systems

Az elosztott verziókezelő rendszerekekben (Distributed Version Control Systems, DVCS) a kliensek nem csak a legfrissebb képet látják, hanem a teljes repositoryt tükrözik, beleértve a teljes előzményt is. Ezáltal, ha akármelyik szerver netán felmondja a szolgálatot és a kliensek ennek a szerver segítségével kommunikáltak egymással, akkor akármelyik kliens vissza tudja másolni a szerverre a repositoryt azért, hogy visszaállítsa.

A képen szöveg, képernyőkép, Betűtípus, diagram látható

Automatikusan generált leírás

3. ábra – Distributed Version Control Systems

# Git

## 1. Letöltés és telepítés

A Git-et a saját weboldaláról lehet letölteni, amit a következő linken érhetsz el:

<https://git-scm.com/downloads>

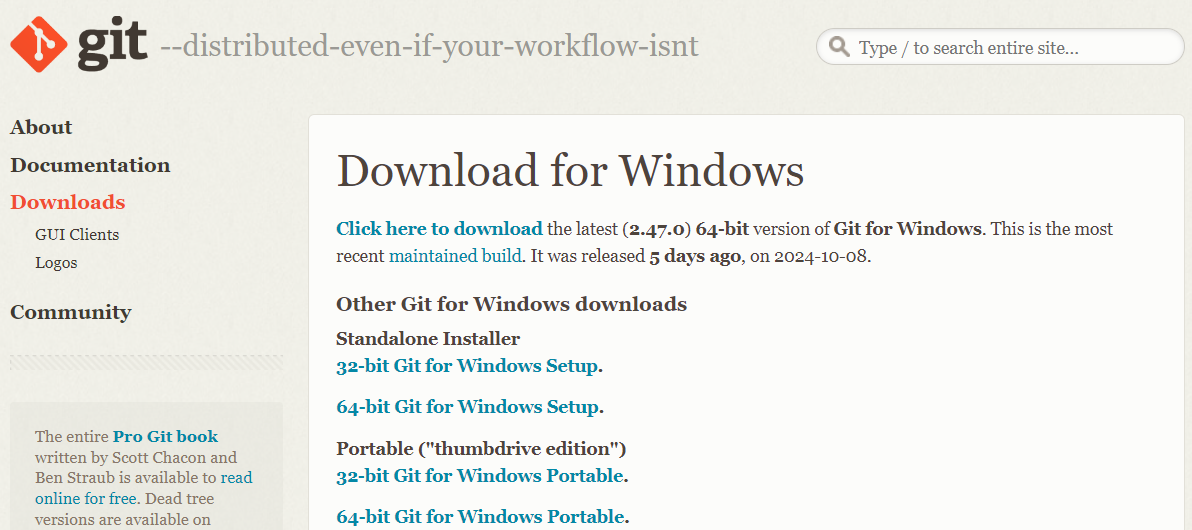
A weboldal egyébként az 4. ábra szerint néz ki. Az oldalon ki tudod választani a saját operációs rendszeredet és aszerint feltelepíteni ezt a szolgáltatást.

A képen szöveg, elektronika, számítógép, képernyőkép látható

Automatikusan generált leírás

4. ábra – Git weboldala

Ha Windowsra szeretnél telepíteni, akkor a letöltések (Downloads) közül a Windowst kell kiválasztanod, majd kattints vagy a 64-bit for Windows Setupra vagy a 32-bit Git for Windows Setup csomagra a Standalone Installer közül. Ezek után egy már jól ismert telepítő varázsló fog megnyílni a letöltés után, amint elindítod a .exe fájlt.



5. ábra – Telepítés Windowsra

A varázslóban először, mint minden szoftvernél a license jelenik, miután erősen tanulmányoztad kattints a Next gombra.

A képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírás

6. ábra- Git installer license

Majd a programnak válaszd ki azt a helyet a számítógépeden, ahova telepíteni szeretnéd.

A képen szöveg, képernyőkép, szoftver, Számítógépes ikon látható

Automatikusan generált leírás

7. ábra – Git installer destination

Ezek után komponenseket tudsz még az alapokon kívűl kijelölni. Én ezt alapbeállításon hagytam, mivel nem volt számomra releváns komponens.

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás

8. ábra – Git installer components

A következő beállításokat nem fogom részletezni, ha valamit nem értesz vagy meg szeretnéd ismerni, hogy mit csinál keress utána az Internet kifürkészhetetlen vad sötét mélyében.

Lesz egy fül, ahol az alap editort tudod kiválasztani a Git számára. Annak, aki nem teljesen van tisztában a Vim használatával ajánlatos egy másik editort választania, viszont, ha szereted a kihívásokat, akkor ne félj a kalandoktól, fedezd fel a Vim-et.

|  |  |
| --- | --- |
| Világító villanykörte körvonalas | **Tipp!**  Csak a Next gombot kell nyomni, amíg el nem érsz egy Install gombhoz, amivel szerintem tudod mit kell csinálnod….kattints rá…. |

Amint feltelepült a program, egy GitBash nevű programot kell keresned a számítógépeden, amellyel el tudod indítani ezt a szuper programot.

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírásA képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás

9. ábra – Git installer start menu folder & default editor

A képen szöveg, képernyőkép, Weblap, Betűtípus látható

Automatikusan generált leírásA képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás10. ábra – Git installer name of the initial branch & PATH environment

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás

11. ábra – Git installer SSH & HTTPS transport

A képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírásA képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírás

12. ábra – Git installer line endings & terminal

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírásA képen szöveg, képernyőkép, szoftver, Betűtípus látható

Automatikusan generált leírás

13. ábra – Git installer behavior of git pull & credential helper

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírás

14. ábra – Git installer extra options

## 2. Projekt létrehozása

### 2.1. git init

A git init parancs egy üres Git tárolót készít vagy újra inicializál egy már meglévőt. Magyarul ezzel hozod létre magát a Git-et egy mappán belül, ezek után tudja a Git a verziókezeléses munkáját elkezdni.

Ez egy .git mappát hoz létre, amiben több almappa is található a Git program számára, ha nem értesz hozzá és nem akarsz napokat szórakozni, ha el állítasz valamit, akkor inkább ne változtass benne semmit…én szóltam…

git init

#### Vizuálisan

A saját környezetemen létrehoztam egy test\_git mappát a D:/ meghajtómon, majd ebbe a mappába beléptem a GitBash nevű program segítségével.

Ahhoz, hogy ezt elérjem a GitBash nevű programmal a következő parancsokat használtam:

Átváltottam a d:/ meghajtóra.

cd d:/

Majd létrehoztam egy test\_git mappát.

mkdir test\_git

Beléptem ebbe a mappába.

cd test\_git

Majd kiadtam a git init parancsot, hogy a Git lássa ezt a mappát.

git init

A GitBashbe a 15.ábra mutatja a parancsokat sorba.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

15. ábra – GitBashbe a git init parancs

## 3. Követem vagy nem követem?

### 3.1. git status

A git status paranccsal meg tudod vizsgálni azt, hogy melyik állományt követi a Git és melyiket nem, illetve azt is látod, hogy mi van hozzáadva a stage-re és mi nem.

Ha nem érted még az előző mondatot, nem baj mindjárt részletezem mi mit jelent.

git status

#### Vizuálisan

Ha most adod ki a git status parancsot a GitBashben, akkor egy hasonló üzenetet kell, hogy kapj válaszként.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

16. ábra – git status üres mappában

Viszont, amint létrehozunk egy fájlt, esetünkben egy alma.txt állományt csináltam, majd kiadtam újra a git status parancsot és máris érdekesebbé vált a helyzet.

A képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírás

17. ábra – git status állománnyal a mappában

Láthatjuk a 17.ábrán, hogy létrehoztunk egy alma.txt-t, majd kiadtuk a git status utasítást. Ezek után láthatjuk, hogy van egy alma.txt fájl az Untracked files (állományok, amelyeket nem követ a Git) között. Ezt a Git egy másik színnel is jelöli, esetünkben pirossal.

### 3.2. git add

A git add paranccsal fel lehet venni adott fájlt vagy fájlokat a Git azon állományai közé, amelyeknek követni fogja a verzióját.

git add <fájl>

Erre úgy kell gondolnod, mintha például lenne egy színház egy üres színpaddal, viszont a nézőtér tele van emberrel, nálunk a nézőtér a test\_git mappa, az embereink pedig csak az alma.txt, hát mit ne mondjak még nem jöttek el sokan az előadásra. Azonban Te, mint színház igazgató felhívod a színpadra az alma.txt-t és innentől kezdve a színpadon lévő emberrel általában mit csinál a közönség? Ugye elkezdik követni a szemével. A Git is ugyanezt teszi a mi esetünkben.

Természetesen létezik könnyítés az utasításhoz, hogy ne kelljen minden fájt egyesével beírni, hogy felkerüljön a színpadra. Van ugyanis egy univerzális mesteri karakter a (.), amit, ha az add után teszünk, minden fájl, ami a mappában található felkerül a színpadra, beleértve a fájlokat, mappákat, almappákat stb. Így néz ki a parancs ezzel az ördögi karakterrel:

git add .

#### Vizuálisan

Szóval, ha kiadjuk a git add parancsot a következő képpen:

git add alma.txt

Majd megnézzük, hogy mit követ a Git vagy úgy is mondhatnám ellenőrizzük, hogy mi változott a mappánkban.

git status

Akkor egy ilyen helyzettel állunk szembe, amelyet a 18.ábra mutat.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

18. ábra – git add utasítás

Láthatjuk, hogy hozzáadtuk az alma txt-t, majd kiadtuk a git status parancsot és máris azok közé a fájlok közé tartozik az alma.txt, amelyeket a Git követ. Vizsgáljuk meg és vegyük észre, hogy a képen már más színnel van jelölve az a fájl, amelyről a Git verzió kezelést csinál.

### 3.3. git rm –cached

A git rm –cached utasítással, azt érjük el, hogy lehívjuk a színpadról az adott állományokat, amelyek a parancs végén szerepelnek.

git rm –-cached <fájl>

#### Vizuálisan

Amint kiadom a parancsot, máris nem fogja követni a Git az adott fájlt.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

19. ábra – git rm

Látod visszament az alma.txt a nézők közé.

## 4. Commitok

A commit egy Git tárolóban pillanatképet készít az összes olyan állományról, ami úgymond trackelve (követve) van a könyvtárban. Ez lényegében egy nagy copy-paste, csak jobb.

A Git úgy tárolja a commitokat, hogy az a lehető legkisebb legyen, ezért, amikor tudja a commitot változások halmazaként, vagy „delta”-ként tömöríti a repository (tároló) egyik verziójájából a másikba.

Fenttart egy log-ot (előzményt), hogy melyik commit mikor készült. Ezért lehet olyan, hogy egy commit felett előd commitok vannak, ezek nyilakkal lesznek jelölve a vizualizációban.

A commitok között lehet váltani is, ami nagyon gyorsan történik.

Az alap parancs a következőképpen néz ki:

git commit

Ez még kiegészül még pl. a -m opcióval. Ezáltal megjegyzést tudsz és kell is írnod egyes commitokhoz.

git commit -m „megjegyzésem”

### Vizuálisan

|  |  |  |
| --- | --- | --- |
| A jobboldalon egy kisebb Git repository található. Két commit van benne eddig. Az első a kezdeti commit C0 és van egy másik utána C1. | | A képen Grafika, rajzfilm, pink, szív látható  Automatikusan generált leírás |
|  | Ha kiadunk egy újabb parancsot:  git commit  Akkor egy új commit fog megjelenni, itt C2 néven. A commitnak, amit éppen csináltunk van egy szülője a C1. | |

GitBashben a valóságban egy commit úgy néz ki, hogy egy -m opcióval ki kell egészíteni a parancsot, ami után, például idézőjelek közé valamilyen megjegyzést írunk a változtatásainkról, így amikor visszaszeretnénk menni a commitokban egyszerűbben megtaláljunk a számunkra megfelelőt.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

20. ábra - git commit

## 5. Előzmények

### 5.1. git log

A git log paranccsal kitudjuk íratni a képernyőre a commitok teljes előzményét. Ezzel látjuk a commit hashét, branchet, a HEAD-et, a készítőt (névvel, emailel, illetve egyéb beállításokkal, ha megadjuk azokat), a dátumot, hogy mikor készült a commit, illetve látjuk a leírást is.

git log

Létezik a parancsnak egy hosszabb verziója, amely azonban lerövidíti az előzményeket, így nem fogjuk látni a teljes leírást és a commithoz tartozó finomságokat, viszont átláthatóbb lesz szerintem a commitok előzménye, mivel egy rövidített hash-t és egy rövid leíást jelenít meg a képernyőn.

Ez a következő utasítással tehetjük meg:

git log --oneline

#### Vizuálisan

Amint kiadjuk az utasítást a 21.ábrához hasonló eredményt fogunk látni.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

21. ábra – git log

A rövidített változat valahogy a 22.ábra szerint fog kinézni.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

22. ábra – git log -oneline

## 6. Branchek

A branchek a Git-ben szintén kis helyeket foglalnak el. Ezek lényegében mutatók (pointers) egy bizonyos commitra. Ezért szokták mondani, hogy „kezdj branchelni korán és sokszor”.

Mivel nem foglal sok helyet, ezért ezzel könnyedén fel lehet logikailag osztani a munkát, amin épp dolgozol. A commitok és a branchek összetartoznak, de ez mindjárt ki is derül, hogy hogyan.

Annyit kell most megjegyezni, hogy amikor egy branchet létrehozol az azt mondja a gépnek, hogy „Minden commitot és szülő commitot szeretnék tartalmazni, ennek a munkának.”

Az alap parancs a következő:

git branch <név>

Ahhoz, hogy az adott branchbe átlépjünk a

git checkout <név>

prancsot kell használni.

A lusták, akik a két parancsot egybe akarják végrehajtani a következő módon tehetik meg:

git checkout -b <név>

Ha elfelejtettük volna, hogy milyen branchek léteznek szimplán írjuk be a git branch utasítást, úgy, hogy semmilyen név nem szerepel utána.

Ez ki listázza a brancheket.

git branch

### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Itt fogunk létrehozni egy új branchet newImage néven a követező paranccsal: git branch newImage | |  |
|  | Most, így a newImage a C1-es commitra hivatkozik és tartalmazza az előzőket is.  Ha a C0 commiton csináltam volna a branchet csak azt tartalmazná a C1-et, ami utána van már nem. | |
| Viszont, ha most commitolunk, akkor a main branch mozog, viszont a newImage nem. Ez azért van, mivel nem a newImage branchben álltunk ezt a (\*) jelöli. | |  |
|  | A Git-nek meg kell mondanunk, hogy branchet szeretnénk váltani, ezt a git checkout <név> parancs kiadásával tudjuk megtenni. Ez az új branchbe fog minket állítani miután commitolunk.  git checkout newImage;  git commit | |

GitBashben így néznek ki az utasítások:

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

23. ábra – git branchek

Kérdezhetnéd, hogy ez most miért jó? Hát azért, mert ha most egy képet teszek a newImage branchbe akkor az csak ott fog látszódni, míg a masterbe az nem lesz ott, így egy estleges alkalmazás featuret, úgy lehet fejleszteni, hogy annak semmilyen hatása ne legyen egy másik branch commitjaira.

Természetesen, ha beleteszünk valamit a newImage branchbe el is kell commitolni a változásokat, majd csak utána lehet branchet váltani.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

24. ábra – branch különbségek

Látod a newImage branchbe létezik egy kep\_a\_branchekrol.png. Ezt a már ismert utasításokkal értem el. git add és git commit. Amikor átváltottam a master branchbe és azt is kilistáztam, ott már nem látható a kép. A commit előzmények is különböznek, ha megnézzük őket.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

25. ábra – git log newImage

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

26. ábra git log master

Amint látod a newImage branchbe létezik egy commit (c10e6bc), amely a masterbe nincs jelen. Azt is látjuk newImage logjába, hogy hol tart a master branch.

## 7. Branchek és a mergelés

### 7.1. git merge

Most, hogy tudunk commitelni és branchelni, valahogy kombinálni kéne a munkánkat különböző branchekből. Ez az tudja, hogy különböző pl. alkalmazások sajátosságait tudjuk összekombinálni.

Az első megoldás a git merge. A mergelés a Gitben két külünleges commitot hoz létre, aminek két egyedi szülője van. Egy commit két szülővel azt jelenti, hogy az A szülő munkáit és a másik B szülő munkáit is szeretném tartalmazni.

git merge <név>

|  |  |
| --- | --- |
| Felkiáltójel egyszínű kitöltéssel | **Figyelj!**  Oda mergelsz, ahol éppen állsz! |

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Van két branchünk, mindegyiknek van egy egyedi commitja C2 és C3. Ez azt jelenti, hogy egyik branch sem tartalmazza a másik branch commitját. Ahhoz, hogy a main branch lássa a bugFix munkáját, ahhoz mergelni kell.  git merge bugFix | |  |
|  | Így most a main egy olyan commitra mutat, aminek két szülője van.  Figyeljük meg azt, hogy abba a branchbe történt a mergelés, amiben éppen állunk! | |

GitBashben létre hoztam egy bugFix nevű branchet a newImage alapján, majd átléptem a master branchbe és összemergeltem a bugFixet a masterrel. Így már a master is tartalmazza a bugFix kép állományát.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

27. ábra mergelés

### 7.2. git rebase

A másik módja, hogy egyesítsük a branchaket a rebase. A rebase fog egy bizonyos mennyiségű commitot és „kivágja” őket, majd beszúrja azokat valahova.

Amíg ez egy kicsit összekavarhat, addig a rebasenek van egy előnye, ami nem más, hogy egy szép egyenes commit sort kapunk, ami nem ágazik ezernyi felé, így a log szebben követhető.

git rebase <név>

|  |  |
| --- | --- |
| Felkiáltójel egyszínű kitöltéssel | **Figyelj!**  Az adott branchet, ahol állunk mozgatjuk, egy másik branch alá, amit a <név> helyére írunk. |

#### Vizuálisan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Megint van két branchünk. Figyelj arra, hogy most a bugFixben állunk.  A munkánkat úgy szeretnénk mozgatni, hogy úgy tűnjön a main-ből fejlődött, míg valójában egy teljesen másik univerzumban.  git rebase main | | | |  |
|  | | Most a munkánk úgy néz ki, mintha szépen egy sorban fejlődött volna.  Jegyezd meg azt, hogy a C3 commit, még mindig létezik valahol, csak el homályosodott a kis fánkban.  Már csak az a probléma, hogy a main nem frissült. | | |
| Hajtsunk végre egy checkoutot mainra, majd rebaseeljünk a bugFixbe.  git rebase bugFix | | |  | |
|  | Mivel a main csak egy őse volt a bugFixnek, ezért a git csak fogta és előbbre tette a main mutatóját a történtekben vagyis a logban. | | | |

GitBashben csináltam egy ez\_a\_bug\_fix.txt-t, ezt elmentettem, majd átléptem masterba, ahol nem található még meg ez a fájl. Miután rebaseltem a mastert a bugFixre, utána már tartalmazza az ez\_a\_bug\_fix.txt-t a master is.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

28. ábra – git rebase

## 8. Mozgás Gitben

Mielőtt komolyabb parancsokat néznénk a Gitben, előtte fontos, hogy megértsünk különböző módszereket arra, amivel a commitok között tudunk lépdelni.

### 8.1. HEAD

Annak a commitnak a szimbolikus neve HEAD, ami éppen checkoutolva van, ez az a commit, amin éppen dolgozol.

A HEAD mindig a legfrissebb commitra mutat, ami a munkafánkban megjelenik. A legtöbb paracs, ami a munkafán változtat, a HEAD módosításával kezdődik.

Normál esetben a HEAD egy ágra (branchre) mutat. Amikor commitolsz a bugFix állapota megváltozik és ez a változás a HEAD-en keresztül látható.

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Most a gyakorlatban megmutatom a HEAD-et commitolás után.  git checkout C1;  git checkout main; | |  |
|  | A HEAD, mindig a main branchünk alatt bújkált.  git commit;  git checkout C2; | |

##### HEAD leválasztása

|  |  |  |
| --- | --- | --- |
| A HEAD leválasztása szimplán csak azt jelenti, hogy egy commithoz kötjük egy branch helyett.  Változtatás előtt, így néz ki a hozzárendelés:  HEAD -> main -> C1 | |  |
|  | Miután checkoutolunk C1-re, így néz ki a hozzárendelés:  HEAD -> C1 | |

GitBashben kiírattam a rövidebb módon a master logjait, majd checkoutoltam a legelső commitra, ami a 192ad61 hash névre hallgat.

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

29. ábra – git HEAD

### 8.2. Relatív Refs

Ahhoz, hogy Gitben mozogjunk meg kell adni a commit hashét, ami kicsit bonyolult tud lenni. Éles környezetben nem valószínű, hogy szépen lesz vizualizálva a munkafa, így csak a git log parancs segítségével tudjuk megnézni a hasheket. Ezenfelül a hashek sokkal hosszabbak a való életben. A példa kedvéért mutatok egy hasht: fed2da64c0efc5293610bdd892f82a58e8cbc5d8

A fejlesztők, azonban gondoltak a felhasználókra, mivel nem kell beírni a teljes hasht, hanem elég annyi karaktert beírni, amivel egyértelműen meghatározható az a bizonyos hash, amit akarunk választani. Az utóbbi példában, így elég a fed2 karakter sorozatot beírni.

Mivel a meghatározása az egyes commitoknak nem túl kényelmes, ezért a Git-nek vannak úgynevezett relatív refs parancs kiegészítései.

A relaív ref-ek segítségével olyan helyről kezdhetünk lépdelni, amit könnyű megjegyezni (pl.: branchek bugFix vagy a HEAD).

Két egyszerű változatát fogom megmutatni:

* Alkalmanként egy commitot ugrik felfelé ^
* Alkalmanként bizonyos számú commitot ugrik felfelé ~<szám>

#### (^) karakter

|  |  |
| --- | --- |
| Vizsgáljuk meg először a (^) karaktert. Minden egyes alkalommal, amikor egy ref név után illeszted a Git megkeresi a szülőjét annak a bizonyos commitnak.  Ezáltal, ha azt mondod main^, ez a main első szülőjével lesz egyenlő. (C1)  A main^^ pedig az ős szülője a main-nek. (C0)  git checkout main^ |  |

##### HEAD-re hivatkozás

|  |  |  |
| --- | --- | --- |
| Hivatkozhatsz a HEAD-re is, mint relatív ref. | |  |
|  | Ha egymás után használjuk párszor, akkor a C0 commithoz érünk.  git checkout C3;  git checkout HEAD^;  git checkout HEAD^;  git checkout HEAD^; | |

GitBashben is ki lehet próbálni valami hasonló végeredményt kell kapj. Én a példa kedvéért csak egy commitot ugrottam a HEAD-hez viszonnyítva. Ha még egyet ugranék akkor már a legelső commithoz érnék.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

30. ábra – git checkout HEAD^

#### (~) karakter

Szóval nem egy, hanem egyszerre több commitot is szeretnél ugrani. Idő igényes lenne a (^) karaktert x20 kiírni nem? Ezért találták ki a (~) karaktert. Ez bemenetnek egy számot vár, hogy mennyit szeretnél ugrani.

##### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Adjunk egy számot a (~) segítségével és menjünk vissza az időben. | |  |
|  | git checkout HEAD~4  Ennyi és máris visszaugrottunk 4 commitot. | |

GitBashben most a legelső commitra ugrottam a tanultakat használva.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

31. ábra –git checkout HEAD~2

## 9. Branch forcing

Most, hogy már profi vagy relative ref-ből is, használjuk is valami érdekesre.

Az egyik leggyakoribb módja, ahogy használom az-az, hogy brancheket mozgatok ide-oda. Közvetlenül hozzá tudok rendelni egy branchet a -f segítségével egy commithoz valahogy így:

git branch -f main HEAD~3

Ez, így erőszakos módon rákényszeríti a main branchet, hogy a HEAD-hez viszonyítva 3 committal hátrébb küldje a main-t.

|  |  |
| --- | --- |
| Világító villanykörte körvonalas | **Jegyezd meg!**  Egy valós Git környezetben a git branch -f parancs nincs engedélyezve arra a branchre, amit éppen használsz. |

### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Így néz ki valahogy a fánk most. | |  |
|  | Az előző parancsot:  git branch -f main HEAD~3  kiadva, ez lesz az eredmény.  A main most a C1-es commithoz van rendelve. | |

GitBashben visszaálllítottam a mastert a kezdő commitra, így az már nem tartalmazza sem a képet, sem a bugFix fájlt. Figyelj arra, hogy amin a HEAD áll branch azt nem tudod visszaállítani, így előtte át kell lépned egy másik branchre, én a bugFixbe vagyok és a mastert állítom vissza.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

32. ábra – git branch forcing

## 10. Változtatások visszaállítása

Számos módja van annak, hogy visszavonjunk változtatásokat. A visszaállítás két szinten történhet. Az első a stage szint, ahol az állományok arra várnak, hogy commitoljuk őket. A másik pedig egy magasabb szinten történik. Mi most az utóbbit fogjuk vizsgálni.

Két fő módja van a visszaállításnak, az egyik a git reset, a másik a git revert.

### 10.1. git reset

A git reset, úgy állítja vissza a változtatásokat, hogy egy branch hivatkozást mozgat vissza egy korábbi commitra. Olyan, mintha a történelmet írná újra. A git reset, úgy mozgatja vissza a branchet, mintha a commit, ahol vagyunk nem is létezett volna.

git reset <ref>

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Nézzük meg hogyan módosul a fánk. | |  |
|  | Ha kiadjuk a következő parancsot, ezt kapjuk.  git reset HEAD~1  A Git, úgy mozgatta a main branchet C1-re, mintha a C2 commit nem is létezne. | |

GitBashben megjelent egy file, amit el is mentettünk, majd észleltük, hogy file buggos és szeretnénk visszavonni a módosításokat. Ezt git reset –-hard HEAD^ paranccsal tehetjük. Azért tettük oda a –-hard opciót, mert enélkül csak a commit törlődik a logból viszont a file és a változások ott maradnak, viszont a –-hard opcióval visszaáll a file az előző verzióra.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

33. ábra – git reset -hard

### 10.2. git revert

Míg a resetelés szuperül működik a helyi brancheinkkel a saját gépünkön, addig ez a „történet módosítás” a távoli brancheken nem végez változtatást.

git revert

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Abban az esetben, ha a visszaállítással kapcsolatos változtatásokat megszeretnénk osztani másokkal a git revert parncsot kell használni. | |  |
|  | Adjuk ki a parancsot:  git revert HEAD  Miiiii?? Egy új commit került a visszaállítandó commit alá?  Ez azért van mert az új C2’ olyan változtatásokat vezet be, amelyek a valóságban visszaállítja a C2 commit módosításait, így pusholni tutod a változtatásokat másokhoz is. | |

GitBashben bekúszott egy hangya.txt, amit elmentettünk a következő committal, amelyet a 34.ábra részletez.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

34. ábra – git revert előkészület

Majd kiadtam a git revert HEAD parancsot, ami azt csinálja, hogy létrehoz egy új commitott (Revert „AHH itt egy hangya” néven, ezt tudod módosítani egy UI-n keresztül, amit a telepítésnél kiválasztottál alap editornak), annak a commitnak az ellentétjével, ahol még a 34.ábra HEAD-je állt. Mivel a 34.ábra HEAD commitjába került elmentésre a hangya.txt, és mivel arra a commitra hivatkoztunk a reverttel, ezáltal ez a fájl a 35.ábra HEAD commitjában már nem szerepel a hangya.txt.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

35. ábra – git revert eredménye

A revert azért is jobb, mint reset, mivel a resettel teljesen törlődnek a változtatások, így ha estleg mégis szeretnénk használni azt a commitot, amit visszaállítottunk, akkor azt nem tudnánk. Reverttel azonban ott marad a régi commit is, így vissza tudunk menni rá bármikor. Tudod ez olyan, mint amikor a főnök kér valamit, majd meggondolja magát, majd megint meggondolja magát.

## 11. Mozgasd a munkád

Így, hogy a Git alapjait elsajátítottad (commitolás, branchelés, a munkafában való mozgást). Csak ezekkel el tudod végezni a legszükségesebb dolgokat a Gitben, nagyjából a 90%-át a munkádnak.

A fennmaradó 10%, azonban nagyon hasznos tud lenni egy komolyabb projekt vagy munka alatt. A következő, amit megmutatok az-az, hogy hogyan tudod a munkád ide-oda mozgatni. Ez azt jelenti, hogy ezt a featuret innen, a másikat onnan, a harmadikat meg amonnan szeretném látni.

### 11.1. git cherry-pick

A legelső parancs a git cherry-pick. A következő képpen néz ki a szintaktikája:

git cherry-pick <commit\_1> <commit\_2> <…>

Szimplán azt csinálja a parancs, hogy bizonyos commitokat másol az aktuális helyünk alá (HEAD).

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Mondjuk azt, hogy van egy repositorynk és van már pár branchünk is benne a side, amit a mainbe szeretnénk másolni. Ezt eltudnánk érni a rebaselve is, de nézzük mit csinál a cherry-pick. | |  |
|  | A következő a parancs:  git cherry-pick C2 C4  Azt értük el vele, hogy a C2 és C4-es commit most már megtalálható felettünk, így a main már tartalmazza azokat is.  Figyeld meg, hogy a main branchben voltunk a változtatásokkor, így a parancsnak ott lesz hatása, ahol éppen állunk. | |

GitBashben létrehoztam egy cherry-pick nevű branchet, amibe összeválogattam különböző commitokat. Kiválasztottam, hogy legyen benne a newImage branch képe (c10e6bc commit) és szerettem volna a hangya.txt is beletenni. (eec484a commit)

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

36. ábra – cherry-pick

### 11.2. git interactive rebase

A cherry-pick nagyon jó választás, amikor tudod az adott commitok hashét.

Azonban mi történik akkor, amikor nem tudod a hasheket? Szerencsére a Git fejlesztői erre is gondoltak. Ilyenkor jön segítségünkre az interactive rebase. Ez a legjobb módja, hogy commitok sorozatát nézzünk át, amelyekkel a rebaselés történne.

Minden, ami az interactive rebase-t jelenti az csupán egy (-i) a rebase parancsban.

Ha ezt a kiegészítést használod, a Git egy UI-t fog feldobni, amely megmutatja melyik commitok fognak a HEAD alá kerülni. Ez szintén megmutatja commitok hashét és megjegyézét is, amely sokat segít, hogy mi-mi.

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Tegyük fel van egy repositoryink. Kiadjuk a következő parancsot:  git rebase -i HEAD~4;  A parancs megnyit egy UI-t, amely tartalmazza a HEAD előtt lévő 4 commitot és ezekkel tudunk bűvészkedni. | |  |
|  | Amit a UI-ban csináltam az-az, hogy a C4-es commitot nem akarom a munkába, illetve átrendeztem a commitok sorrendjét, így most C2 -> C5 -> C3 a sorrend. | |

GitBaschbe létrehoztam egy cherry-pick\_interactive branchet (37.ábra), majd kiadtam a git rebase -i HEAD~4 utasítást és megnyílt Vim-ben a UI (38.ábra), ami megkönnyíti a munkát.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

37. ábra – interactive branch

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

38. ábra – interactive UI

Valami hasonló UI kell, hogy megjelenjen nálad is. A 38.ábrán láthatod, hogy amelyik commit előtt pick van azt választottam ki, ami előtt drop van, azt pedig kihagytam a választásból.

Lent a kék szövegben a 38.ábrán láthatod, hogy melyik előtag mit csinálna.

A 39.ábrán pedig láthatod a végeredményt.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

39. ábra – interactive rebase végeredmény

## 12. Távoli repository

Távoli repositroyk nem olyan vészesek, igazából ezek csak másolatok a te repository-aidról másik számítógépeken. Tipikusan, képes vagy arra, hogy ezekkel a számítógépekkel kommunikálj az Internet segítségével, így commitokat tudnak egymásnak küldeni.

Távoli repositoryk néhány előnye:

* Először is egy nagyon jó backupnak szolgál. A lokális Git repositoryk képesek visszaállítani az állományokat egy korábbi állapotra, azonban az összes információ helyileg van tárolva. Távoli repok segítségével nyugodtan elvesztheted a munkád, mivel azok segítségével vissza tudod állítani a munkád oda, ahol éppen abbahagytad azt.
* Másodszor, tudsz másokkal kódolni, így nem egyedül kell szidnod a kódot, hogy miért nem megy úgy, ahogyan azt szeretnéd. A barátaidnak is megtudod könnyedén mutatni a kódodat, sőt be is tudnak kapsolódni a fejlesztésedbe.

Ma már egyre népszerűbbé válik olyan weboldalak használata (pl. GitHub), amelyek vizualizálják az eseményeket a távoli repokban.

Eddig a pontig arra fókuszáltam, hogy a lokális repositroyban otthon érezd magad, de most ki kell lépni ebből a szobából.

|  |  |
| --- | --- |
| Követés leállítása kitűző egyszínű kitöltéssel | **Figyelem!**  A következőkben már nem fogsz látni GitBash példát, mert úgy gondolom most már eleget tudsz, ahhoz, hogyTe gyere rá egy esetleges problémára. Ha valami nem megy keress rá az Interneten vagy nézd meg az ajánlásokba a videókat. |

### 12.1. git clone

Ahhoz, hogy elkezdhessük a tanulás a git clone parancsra lesz szükségünk.

Technikailag ez a parancs lesz hasznunkra, amikor helyi másolatokat szeretnél készíteni távoli repokról.

git clone <url>

Készíthetsz ennek a dokumentációnak a repojáról is egy másolatot akár a példa kedvéért.

Ha az url helyére beilleszted a következőt, akkor le clonozod a távoli repomat:

<https://github.com/BalazsGyulai/Git_Documentation.git>

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Van egy távoli reponk, amely nagyon megtetszett vagy hasonló. Az a lényeg szeretnénk módosítgatni a mi saját számítógépünkön. | |  |
|  | Ha kiadjuk a parancsot:  git clone <url>  Akkor máris kapunk egy másolatota távoli reporól (fehér szagatott szélű). | |

### 12.2. Távoli branchek

Mint már feltűnhetett van némi változás a reponkban. Hozzuk az előző példa végeredményét:

A képen képernyőkép, pink, Grafika, Magenta látható

Automatikusan generált leírás

Van a távoli reponk (Fehér szagatott szegéllyel jobb oldalt). Ebben van egy main branch.

A bal oldalon pedig szintén van egy main branch, de van egy o/main branch is. Ez a o/main jelképezi a távoli branchet a jobb oldalt.

Azért van külön branch és jelkép, hogy könnyen meg tudd különböztetni, hogy mi van a helyi branchben és mi van a távolin. Ennek főleg akkor van jelentősége, amikor másokkal dolgozol együtt.

Amikor checkoutolsz egy távoli repo helyzetére, akkor a HEAD le lesz választva onnan, ahol éppen dolgozol. Ez azért van így, mert ezekben a távoli branchekben csak akkor fogod látni a változtatásaid, ha azokat meg is osztod másokkal.

Mit jelent az o/?

Két része van ennek a jelölésnek. A megjelenése pedig két dologtól függ:

<remote név>/<branch név>

Ha megnézed a példát o/main, a branch név a main és a távoli reponak a neve o.

A legtöbb fejlesztő a távoli mainjüket originnek nevezik nem pedig o-nak. Ez az elnevezés olyan népszerű, hogy a git alapból originnek nevezi a távolit maint, amikor klónozol.

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Vizsgáljunk meg egy távoli branchet. | |  |
|  | Adjuk ki a következő parancsot:  git checkout o/main;  git commit;  Ahogy látjuk a git leválasztotta HEAD-et és nem frissítette az o/main-t, amikor egy új commitot hoztunk létre. | |

### 12.3. git fetch

A következőkben megtanuljuk, hogyan lehet távolról adatokat lehívni. Ehhez a git fetch parancsot fogjuk használni.

git fetch

A git fetch két fő dolgot hajt végre:

* Letölti azokat a commitokat, amelyek a távoli repon megtalálhatóak és…
* frissíti a helyi branch állapotát. Esetünkben ez a o/main

Igazából ez a parancs szinkronba hozza a munkánkat a távoli repoval.

A git fetch az interneten kommunikál a távoli repoval például https:// vagy git:// protokollokat használva.

Fontos megjegyezni, hogy a parancs nem végez el minden változtatást a helyi számítógépünkön. Nem fogja frissíteni vagy változtatni a main branchedet vagy az állományok kinézetét a gépeden. Sokan azt hiszik, hogy ha lefuttatják a git fetch parancsot, akkor a helyi munkájuk a távoli állapotokat fogja mutatni. Erre majd később fogunk tanulni egy parancsot, ami pontosan fogja ezt csinálja.

Viszont most csak annyit kell tudni, hogy a git fetch az szimplán csak egy letöltési folyamat.

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Van egy távoli branch, de az már előbbre tart, mint a mi helyi reponk. Nézzük, hogyan tudjuk ezt a problémát megoldani. | |  |
|  | Kiadjuk a parancsot  git fetch  Máris látjuk, hogy a helyi számítógépen is megtörténtek a változások a tanultak alapján. | |

### 12.4. git pull

A távoli állapotra való frissítés annyira gyakori, hogy a Git kifejlesztett egy parancsot, ami nem csak letölti, hanem mergeli is a változásokat. Ez a parancs a git pull.

git pull

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Mi történik a git pull közben? | |  |
|  | A parancs  git pull  Lényegében két legyet üt egy csapásra, mivel nem kell git fetch, majd git merge parancsot futatnunk. | |

### 12.5. git remote

A git remote paranccsal be tudsz állítani egy távoli repot célnak, abban az esetben, ha nem clonoztál, hanem mondjuk elkezdtél lokálisan dolgozgatni, viszont annyira megnőtt a rendszered, hogy szeretnél például egy backupot csinálni egy felhőbe.

Az alap utasítás így néz ki:

git remote set-url origin <url>

Az url helyére ssh-t is állíthatsz, nem fontos https-t.

Ellenőrizni is tudod, hogy valóban beállítottad-e célt az alábbi utasítással

git remove -v

A kimenetben két sort kell látnod egyet fetchre egyet pushra.

### 12.6. git config

A config parancsokkal a felhasználó nevedet, email címedet stb. tudod beállítani, ezek szerepelnek a commitok mellett, ezért, ha másokkal dolgozol értelmes neveket használj, hogy tudják ki vagy.

Felhasználónév beállítást ezzel tudod megtenni:

git config –-global user.name „neved”

Email címet pedig ezzel tudsz beállítani:

git config –-global user.email „email”

### 12.6. git push

Szóval most már tudunk letölteni, de mi van azzal a munkával, amit én végeztem, azt mikor fogják látni a többiek?

A git pull-nak az ellentéte a git push.

git push

Ez felel azért, hogy bizonyos változtatások a commitjaidon felkerüljön a távoli repoba.

#### Vizuálisan

|  |  |  |
| --- | --- | --- |
| Nézd csináltam egy változtatást a távoli repohoz képest, de ott ez még nincs meg. Hogy tudom a változtatásomat megosztani? | |  |
|  | Lefutattom a  git push  parancsot és máris frissül a távoli repo is. | |

### 12.7. Verzió eltérés

Képzeld el, hogy egy vagy többen is dolgoztok egy távoli repon. Mondjuk te kiveszel egy hét szabit, egy csodálatos meleg nyári hétfőn. Mire visszamész dolgozni, addigra már a munkatársaid több változtatást is megosztottak egymás között, így az ő helyi repojuk a legfrissebb verziószámmal rendelkezik, viszont te is folytattad a munkádat, amikor visszamentél dolgozni és megszeretnéd azt osztani.

Viszont, ha a git push parancsot használod, akkor most a távoli reponak visszakéne állítani magát egy régebbi verzióra? Megkéne próbálnia hozzáadni a te változtatásaidat miközben nem távolít el semmilyen újabb dolgot a fájlokból? Vagy szimplán a te munkádat kéne eldobnia, mondván a tiéd az régebbi, így az már nem releváns neki?

Egy ilyen szituációban a Git nem engedi, hogy pusholj. Igazából rákényszerít téged, hogy frissíts a legújabb változásokra, mielőtt megpróbálod megosztani a munkád.

A legegyszerűbb módja ennek, ha rebase-el frissíted a munkádat. Nézzük, hogyan teheted ezt meg.

#### Vizuálisan

##### git rebase

|  |  |
| --- | --- |
| Most rebaseljünk, mielőtt pusholunk… |  |
|  | Lefutatjuk a következő parancsokat:  git fetch;  git rebase o/main;  git push;  Ennyi! Frissítettük a helyi reponkat, majd megosztottuk azt. |

##### git merge

|  |  |  |
| --- | --- | --- |
| Természetesen mergel is megoldhatjuk a problémát. | |  |
|  | Parancsok:  git fetch;  git merge o/main;  git push;  Ezzel frissítettük a helyi megjelenését a távoli reponak, majd mergeltük az új munkákat a mi munkánkba, majd megosztottuk azt. | |

##### git pull –rebase

|  |  |  |
| --- | --- | --- |
| Természetesen ennek is van könnyebb módja, ami csak két sorba kerül. | |  |
|  | Parancs:  git pull –rebase;  git push;  Ugyanaz, mint előbb csak rövidebben. | |

##### git pull

|  |  |  |
| --- | --- | --- |
| És most lássuk sima pullal. | |  |
|  | Parancsok:  git pull;  git push;  Újra, ugyanaz, mint előbb. | |

## 13. Távoli repo frissítés eldobás

Ha egy nagy projekten dolgozol sok csapattal, nagy valószínűséggel Pull Requestet igényel az, hogy mergeld a munkádat. Amennyibben commitoltál a helyi main alá és megpróbálsz pusholni egy hasonló üzenet üdvözölhet:

! [remote rejected] main -> main (TF402455: Pushes to this branch are not permitted; you must use a pull request to update this branch.)

### 13.1. Miért dobta el?

Azért dobta el a commit pusholást, mivel a main szabályzat szerint pull requestet kell használni.

A következő lépéseket kell végrehajtanod, készítesz egy branchet, majd pusholod és egy pull requestet csinálsz, viszont elfelejtetted és közvetlenül mainra commiteltél. Most pedig nem tudod pusholni a változtatásaid.

### 13.2. Megoldás

Csinálj egy másik branchet pl. feature és pushold azt a távoli repoba. Majd hajts végre egy resetet a mainen, hogy szinkronba legyen a távoli repoval, különben egyéb problémák léphetnek fel a következő alkalommal, amikor pullolsz és valaki másnak a commitja konfliktusba kerül a tiéddel.

# Források és ajánló

Ez a dokumentáció tanító szándékkal jött létre és szabadon felhasználható!

A dokumentációban saját, illetve mások által létrehozott vizualizációkat tartalmaz. Szerepelnek ábrák, amik alatt nincsen ábra jelölés, ez egyszerűen azért van, mert azok nem tőlem származnak, hanem egy online weboldalon találhatóak egy játékból, ami lentebb megtalálható. A leírás, ami a dokumentációban van nagyban hasonlít az ugyanazon az oldalon szereplő leírásokhoz, mivel azt vettem alapnak. A dokumentáció tartalmaz még más forrásokat is, mint például weboldalak, YouTube sorozat, ProGit könyv, amiket érdemes elolvasni, megnézni vagy végigjátszani.

A következő linkek, mind a könyv forrásai, amiket ajánlok ahogy a fentiekben említettem:

<https://www.atlassian.com/git>

<https://kodekloud.com/blog/change-remote-origin-in-git/>

Játék:

<https://learngitbranching.js.org/>

[SanFranciscobol Jottem](https://sanfranciscoboljottem.com/) YouTube sorozat:

<https://sanfranciscoboljottem.com/>

<https://youtu.be/XDKZu9kuEn8?si=9vdDZNfwELDMKeh7>

<https://youtu.be/_DpBYNQwQmU?si=HU0XgiVwxFx139LI>

<https://youtu.be/nOmw00Yh_to?si=sKL5-gSir3QjOrcH>

<https://youtu.be/KqdZ1B8sS8k?si=To1-TtadknVwm7B0>

A Git saját dokumentációjában még több parancs található, és a dokumentációban szereplő utasításokhoz is találhatók még kiegészítések.

<https://git-scm.com/docs>

Annak, aki még mélyebben szeretné megismerni a Gitet, ajánlom a Pro Git nevű könyvet.

<https://git-scm.com/book/en/v2>

Ajánlok még egy cheat sheetet, amiben a dokumentációban szereplő parancsok szerepelnek. Érdemes egy pillantást vetni hátha egyszer még hasznos lehet.

<https://training.github.com/>

# Ábrajegyzék

[1. ábra – Local Version Control Systems 0](#_Toc179738240)

[2. ábra – Centralized Version Control Systems 1](#_Toc179738241)

[3. ábra – Distributed Version Control Systems 1](#_Toc179738242)

[4. ábra – Git weboldala 0](#_Toc179738243)

[5. ábra – Telepítés Windowsra 0](#_Toc179738244)

[6. ábra- Git installer license 1](#_Toc179738245)

[7. ábra – Git installer destination 1](#_Toc179738246)

[8. ábra – Git installer components 2](#_Toc179738247)

[9. ábra – Git installer start menu folder & default editor 3](#_Toc179738248)

[10. ábra – Git installer name of the initial branch & PATH environment 3](#_Toc179738249)

[11. ábra – Git installer SSH & HTTPS transport 3](#_Toc179738250)

[12. ábra – Git installer line endings & terminal 4](#_Toc179738251)

[13. ábra – Git installer behavior of git pull & credential helper 4](#_Toc179738252)

[14. ábra – Git installer extra options 4](#_Toc179738253)

[15. ábra – GitBashbe a git init parancs 5](#_Toc179738254)

[16. ábra – git status üres mappában 6](#_Toc179738255)

[17. ábra – git status állománnyal a mappában 6](#_Toc179738256)

[18. ábra – git add utasítás 7](#_Toc179738257)

[19. ábra – git rm 8](#_Toc179738258)

[20. ábra - git commit 10](#_Toc179738259)

[21. ábra – git log 11](#_Toc179738260)

[22. ábra – git log -oneline 11](#_Toc179738261)

[23. ábra – git branchek 13](#_Toc179738262)

[24. ábra – branch különbségek 14](#_Toc179738263)

[25. ábra – git log newImage 14](#_Toc179738264)

[26. ábra git log master 14](#_Toc179738265)

[27. ábra mergelés 16](#_Toc179738266)

[28. ábra – git rebase 18](#_Toc179738267)

[29. ábra – git HEAD 20](#_Toc179738268)

[30. ábra – git checkout HEAD^ 22](#_Toc179738269)

[31. ábra –git checkout HEAD~2 23](#_Toc179738270)

[32. ábra – git branch forcing 25](#_Toc179738271)

[33. ábra – git reset -hard 27](#_Toc179738272)

[34. ábra – git revert előkészület 29](#_Toc179738273)

[35. ábra – git revert eredménye 29](#_Toc179738274)

[36. ábra – cherry-pick 31](#_Toc179738275)

[37. ábra – interactive branch 32](#_Toc179738276)

[38. ábra – interactive UI 33](#_Toc179738277)

[39. ábra – interactive rebase végeredmény 33](#_Toc179738278)