Verziókezelő rendszer – GIT

Tartalomjegyzék

[REPOSITORY 1](#_Toc146115356)

[Szervezett tesztek 1](#_Toc146115357)

[Verziókezelő rendszerek struktúrája 2](#_Toc146115358)

[GIT 3](#_Toc146115359)

[GIT BASH 3](#_Toc146115360)

[Távoli repo-hoz csatlakozás 4](#_Toc146115361)

[Távoli repo clone-ozása 5](#_Toc146115362)

[Repository-k állapotok vizsgálata, módosítása 6](#_Toc146115363)

[Törlések: 6](#_Toc146115364)

[Branch 6](#_Toc146115365)

[Branch flow (hierarchia) 7](#_Toc146115366)

[Git repository-k 8](#_Toc146115367)

[Git használatának összefoglalója 8](#_Toc146115368)

[Fogalmak tisztázása 8](#_Toc146115369)

[Felmerülő kérdések 9](#_Toc146115370)

[Forrás 9](#_Toc146115371)

# REPOSITORY

Központi kódtároló: virtuális raktár/könyvtár (tárol, rendszerez, hozzáférést biztosít)

Ebben lévő file az egy verzió. Másvalaki feltölti (commit-olja) ezt a file-t módosítva, így a verziókezelő ezt a file-t verziózza, így az eredeti is megmarad a verziózás miatt.

A verziózással nyomon követhető a módosítások ideje, dátuma, neve, ki commit-olta, azaz módosította és visszaugorhatunk egy korábbi változatára a file-nak.

# Szervezett tesztek

Kódszinten és programszinten történő tesztelés, egy-egy módosítást követően.

Agilis szoftverfejlesztés esetén mondjuk 2 hét után az elkészült módosításokat commit-olják, azaz felküldik és tesztelik.

**UNIT TEST**: a szgép kipróbálja, hogy a kód részletek helyesek-e

**AUTOMATIZÁLT tesztek**: a szgép egy scripten végighaladva kipróbálja, hogy működnek-e a program funkciói.

**FUNKCIONÁLIS teszt**: a QA (Quality Assurance: minőség ellenőrök) teszterek kipróbálják a programot, hogy felhasználói oldalról minden rendben van-e

# Verziókezelő rendszerek struktúrája

SVN: Központosított verziókezelő = Centralized version control system (**CVCS**) vs.

GIT: Megosztott verziókezelő = Distributed version control system (**DVCS**)

* Mindkettő más, nézőpont és igény kérdése dönti el, melyiket használják.

Szerver a könyvtár SVN esetében ez a központ, aminek küldjük az adatot. Módosítjuk a file-t, a szerver regisztrálja a commit-ot, azaz amit beküldtem, a végleges módosításomat. Mehet a könyvtárba és ott el van tárolva. Más lekéri a legutolsó módosított verziót. Ezért kell a repo-t rendszeresen frissíteni, gyakran kell egy update request-et küldeni a repo felé, hogy mindig a legfrissebb verzió kerüljön hozzájuk (git-et használók egy pull-t küldenek). Ehhez összeköttetés kell a szerverrel. Mindenki a repo-hoz csatlakozik, az van középen.

GIT esetében, azaz megosztott (DVCS) verziókezelők esetében abban más a helyzet, hogy ugyanúgy középen van a szerver, de valójában mi nem függünk tőle, mert minden egyes felhasználó létrehoz a helyi gépén egy lokális repository-t (tárolót). Ennek nagy előnye, hogy nem kell folyamatos internetkapcsolat ahhoz, hogy a file-okon dolgozzunk. Bármilyen módosítást csinálok, először a saját repo-mnak küldöm el a módosítást (regisztrálja be a változtatást), amihez nem kell internetes kapcsolat. Központi szerver erről még nem tud, csak később, amikor egy **PUSH** request-et küldök (jelzem a központi szervernek, hogy az én helyi repo-m módosult).

**CONFLICT** akkor keletkezik, amikor egy egymástól eltérő módosítás történt ugyanazon a file-on PUSH-olás alkalmával. Ezért szükséges a gyakori szinkronizálás a központi és a helyi repo-k között.

# GIT

Technológia, nem egyenlő a GITHUB-bal, ami egy rendszer és ezt a technológiát használják.

<https://git-scm.com/download/win>

cmd-be **systeminfo** segítségével megtudjuk, hogy 32 vagy 64 bites verziót kell telepíteni.

# GIT BASH

|  |  |
| --- | --- |
| cls | cmd ablak tisztítása |
| dir | adott könyvtár kilistázása |
| cd | belépés |
| cd .. | visszalépés |
| mkdir | mappa létrehozása |
| touch readme.txt | readme.txt file létrehozása |
| status | info-t ad vissza az adott mappáról |
| git **init** | új git repository létrehozása (inicializálása) az adott mappán belül |
| git **status** | adott repo státuszát nézzük meg |
| git **add** readme.txt | inicializált repo-hoz hozzáadja a readme.txt file-t |
| git **commit** -m „message” | repo-ba küldés üzenettel (fontos!) |
|  |  |
| git **clone** <https://github.com/kerekesjuca/Practice.git> | repo klónozása (github-on találhatót a helyi gépre klónozza) |
| git add \*.txt  git add . | adja hozzá az összes txt állományt  mindent adjon hozzá |
| git **log** | commit log-ok |
| git **remote** add origin https://github.com/kerekesjuca/Practice.git | távoli repo hozzáadása a helyi gépünkön |
| git **push** -u origin main | commit-ált csomag felküldése központi szerverre main branch-be |
| git **pull** origin main | távoli repo-ról állomány(ok) beszerzése lokális repo-ra |
| git pull origin main --allow-unrelated-histories | pull paranccsal állományok beszerzése |
| rm -rf ./\* | lokális repo törlése (adott repo mappájában állva!) |
| git **diff** HEAD | különbség két repo között |
| git diff **--staged** | megmutatja a különbségeket azon file-ok között, amik az add paranccsal hozzá vannak adva |
| git diff **--cached** | megmutatja egy file különbségét, amik add-dal hozzá lett adva |
| git **reset** file.kit | commit visszavonása |
| git **checkout --** file.kit | a módosított file-t vissza lehet állítani a módosítás előtti állapotára |
| git **branch** branch\_name | branch létrehozása |
| git **checkout** branch\_name | branch váltása |
| git **rm** ’\*.txt’  git commit -m „Remove all txt files” | branch-ben lévő txt file-ok törlése  törlés utáni commit-álás |
| git **merge** branch\_name | összefésüli a branch\_name nevű branch-ben található fájlokat |
| git **branch -d** branch\_name | branch törlése (d = delete) |
| git log **--graph --all** | log grafikusabb változata |

Logikailag összetartozó dolgokat kell commit-olni. Külön-külön commit-oljuk a módosított file-okat. Minél többször commit-olsz, annál rétegesebben lehet megvizsgálni, ha valahol hiba van.

**git status add** utasítás után kiírja, hogy „changes to be comitted”, amely azt jelenti, hogy „változások, amik commit-olásra várnak”.

Commit-tal a korábban hozzáadott módosításokat tartalmazó „csomag”, amit később push-sal töltjük fel a távoli repo-hoz.

# Távoli repo-hoz csatlakozás

Ehhez először be kell jelentkezni a távoli könyvtárba, amihez pl. egy GitHub account-ot kell létrehozni.

GIT BASH: **git config user.name „username”**

**git config user.password „password”**

A sikeres bejelentkezés után az alábbiak szerint csatlakozhatunk a távoli repo-hoz: **git remote add origin** <https://github.com/kerekesjuca/Practice.git>

* origin egy alias-a az utána megadott path-nak, így később elég az origin szót megadni

Ezután a push paranccsal tudjuk elküldeni a commit-olt csomagot a távoli repo-ra: **git push -u origin main**

* -u kapcsoló teszi lehetővé, hogy a későbbiekben az [origin main] -t ne kelljen minden push-olásnál begépelni.

Távoli repo használatának előnye, hogy másnak is jogot adhatunk rá, hogy távolról rákapcsolódjon erre a repo-ra. Saját gépen beállítjuk a saját GitHub account-unkat.

**git config --global user.name/email „username/github\_email”**

**git config --global user.password „password”**

Amikor az a helyzet adódik, hogy előbb van létrehozva lokálisan repo, mint a távoli szerveren, akkor előfordulhat, hogy a távoli szerveren vannak olyan állományok, amik lokálisan nincsenek meg. Ezen állományokat a pull paranccsal tudjuk beszerezni lokális repo-nkba.

**git pull origin main --allow-unrelated-histories** paranccsal végrehajtható a pull, aztán jöhet a push.

Ha már létre van hozva a központi repository, akkor elég clone-ozni a távoliról lokálisra.

# Távoli repo clone-ozása

**git clone** <https://github.com/kerekesjuca/Practice.git> paranccsal egy távoli repo teljes tartalmát tudjuk lokálisan létrehozni.

# Repository állapotok vizsgálata, módosítása

Ha többen használnak egy távoli repo-t, előfordulhat, hogy a nálunk lévő állományok nem a legfrissebbek, ezért mielőtt push-olunk, előtte kötelező pull-ozni a távoli repo-ról. **git status** parancs kiadása után látható, hogy egy file-t módosítottak-e. Ahhoz, hogy megtudjuk, mi a különbség a két repo között, azt a **git diff** **HEAD** (legutóbbi végleges változatunk) paranccsal tudjuk megnézni.

* Piros: törölt
* Zöld: hozzáadott

Mindig az előző felcommitolt változathoz hasonlítja a jelenlegi állapotot.

**HEAD:** azt az állapotot mutatja, amiben még nem történtek változtatások.

**git diff --staged** megmutatja a különbségeket azon file-ok között, amik add paranccsal hozzá vannak adva.

**git diff --cached** egyetlen file esetén listázza a különbségeket.

**git reset test.txt** paranccsal lehetőség van az add-olt txt fájlt visszavonni.

**git checkout -- test.txt** esetén a módosított file-t vissza lehet állítani a módosítás előtti állapotára.

a Git 2.23-as verziójával a **git checkout** parancs két részre lett osztva:

* visszaállítás: **restore**
* branch váltás: **switch**

# Törlések:

repository: remove

stage-ről: reset

repo-ból: rm

branch: -d

# Branch

A branch-ek különböző színterei egy adott projektnek. Különböző színterek, ahol elkülönítve ezek segítségével történnek a fejlesztések, hibajavítások, release-ek kiadása. Egy fejlesztés során elkerülhetetlen, hogy különböző színtereken történjen a fejlesztésből adódó állományok és stabilan működő alkalmazáshoz tartozó file-ok kezelése.

## Branch flow (hierarchia)

**main** (korábbi neve master): éles, működő alkalmazások és állományai. Itt nem történik fejlesztés. Ezt látják a felhasználók, a főbb verziók itt helyezkednek el.

**develop**: fejlesztői környezet, ahol a fejlesztők elhelyezik az állományaikat. Ha ezek jóváhagyásra kerülnek, akkor engedélyezett módosításokat (commit-okat) átrakják a main branch-be.

**feature branches**: új funkciók kidolgozására szolgáló színtér.

**release branches**: tesztelők számára létrehozott branch.

**hotfixes**: éles rendszer hibáinak javítására szolgáló branch.

**git branch branch\_name** paranccsal hozunk létre másik branch-et.

**git checkout branch\_name** branch váltás branch nevének megadásával.

main branch a hierarchiában a csúcson helyezkedik el. Ezalatt lehet létrehozni más branch-eket. Ahhoz, hogy a main branch-ben láthatóvá váljanak más branch-ek állományai, merge-ölni kell a két branch-et: **git merge branch\_name** (amiben állunk oda másolja be a megadott branch\_name nevű branch file-jait).

git log vezeti a módosításokat, ezért ha valamit elrontunk, vissza tudjuk állítani.

**git branch -d branch\_name** paranccsal törölhető a megadott branch.

# Conflict kezelés

*nem készült még oktató videó a San Fransisco-ból jöttem sorozatán belül.*

# Git repository-k

<https://github.com/>

<https://about.gitlab.com/>: ingyenes privát git repo

<https://beanstalkapp.com/>

# Git használatának összefoglalója

<https://www.youtube.com/watch?v=Qe9Vu98MEWE&list=PLyriihBWoulw1KQ2aM-NMnAoVPZevfmZq&index=9>

# Fogalmak tisztázása

Release

Request

Pull request

Feature

Check out

Deploy

Fetch

Pull

Plane lista

Distributed

Remote

Cherry pick

deploy

# Felmerülő kérdések

1. Hogyan lehet repository-t törölni ? – github oldalon repo-n belül settings és ott delete

# Forrás

<https://www.youtube.com/watch?v=XDKZu9kuEn8&list=PLyriihBWoulw1KQ2aM-NMnAoVPZevfmZq&index=1>