

**GitHub:** internetes szolgáltatás, ami lehetővé teszi, hogy távoli szerveren Git tárolót hozzunk létre, másokkal közös tárolóba dolgozzunk. A GitHub ezen kívül jogosultságkezelést, hibakövető rendszert, wikit is biztosít nekünk, valamint projektmenedzsmentet és a folyamatos integrációt (continuous integration) segítő eszközöket is.

- **Távoli tároló (remote repository, remote):** távoli tárolónak, remote-nak nevezzük azt a tárolót, amit a jelenlegi tárolónk követ (*track*). A távoli tároló általában (de nem feltétlenül) tényleg *távol*, távoli szerveren van, az interneten keresztül érjük el.
- **Eredeti (origin):** hagyományosan annak a távoli tárolónak a neve, amelyből a jelenlegi tárolónkat klónoztuk.
- **Klón (clone):** ha el tudunk érni egy távoli tárolót, akkor azt a *git clone <távoli tároló címe>* paranccsal *klónozni* tudjuk, azaz le tudjuk másolni a gépünkre. Az így a gépünkön létrejött tárolóra az eredeti (origin) tároló klónjaként hivatkozhatunk.
- **Autentikáció:** a távoli tárolók klónozásához gyakran biztonsági rendszerek, hogy illetéktelenek ne férhessenek hozzá. A GitHub két autentikációs, azaz azonosítási módot támogat: a **https** autentikációhoz felhasználónév és jelszó, vagy személyes *token* szükséges; míg az **ssh** autentikációhoz *ssh kulcspár*, illetve publikus kulcsunk regisztrálása. [GitHubos útmutató az SSH kulcsokról \(angolul\)](#)<sup>1</sup>.
- **Push:** a *git push (tolni, nyomni)* parancshasználatával tölthetjük fel (“nyomhatjuk/tolhatjuk fel”) a helyi tárolónk tartalmát egy távoli tárolóba. Ilyenkor nem igazán a tartalmat töltjük fel, hanem azokat a *commit*okat, amelyek a helyi tárolóban megtalálhatóak, a távoli tárolóban pedig nem. Erre a műveletre *pusholás*ként hivatkozunk.
- **Pull:** a push (tolni) ellenpárja a pull (húzni). A *git pull* parancs használatával a távol tároló tartalmát a helyi tárolónkba tudjuk *lehúzni*.
- **“Sync”:** a Visual Studio Code lehetőséget ad, hogy a push és a pull parancsokat mintegy egyszerre kiadva távoli és helyi tárolóink tartalmát *szinkronizáljuk*.

---

<sup>1</sup> Az útmutató nem emeli ki, de fontos, hogy Windows operációs rendszer használatakor a kulcsok létrehozásakor mindig adjunk meg abszolút elérési útvonalat!

**Fetch:** Korábban azt mondtuk, hogy a *git pull* paranccsal tölthetjük le a távoli tároló tartalmát a helyi tárolóba. Ez nem teljesen igaz: a *pull* valójában többet tesz: egyrészt letölti a távoli tároló változtatásait, majd a jelenlegi branchbe olvasztja be (*merge*). A *git fetch* ("elhozni") parancs ennek a folyamatnak az első lépése: letölti a távoli tároló változtatásait, de a helyi tároló brancheihez nem nyúl.

- **Fork:** ha van egy távoli tároló, amelybe nincs jogunk változtatásokat feltölteni, akkor is gyakran van lehetőségünk a tároló lemásolására. Ha ebben a másolatban változásokat hajtunk végre, amelyek az eredetiben nem jelennek meg, azt mondjuk, hogy a projektet *leágaztattuk*, új forkot hoztunk létre. A fork változtatásait az eredeti tárolóban létrehozott *pull kérelmek* kiadásával vezethetjük vissza az eredeti tárolóba. Ez a szóhasználat a nyílt forráskódú világból ered, ahol a fejlesztők gyakran hozták létre saját, leágaztatott változatukat egy projektből, ezt hívták forknak. Néha a fork népszerűbb lett, mint az eredeti!
- **Merge, merging:** azaz *beolvasztás, összeolvasztás*. Ha két ág (branch) eltér egymástól, a *git merge* paranccsal *olvaszthatjuk bele* a másik ág változtatásait a munkakönyvtár állapotába. Ilyenkor tényleg *összeolvasztásról* van szó, hiszen mindkét ág változtatásait igyekszünk megtartani.
- **Merge conflict:** ha két ág egy állománynak ugyanazokat a sorait változtatja, akkor a Git magától nem tudja eldönteni, hogy melyik változtatás a fontosabb, melyik jusson érvényre. Ilyenkor *konfliktusról* beszélünk, amely emberi beavatkozást igényel.
- **Rebase:** azaz átemelés. A *git rebase* paranccsal commitok sorozatát lehet egy commitban összegezni. Arra is alkalmas, hogy egy commit őse más commit legyen, vagyis lényegében egy adag változtatás más alapállapottól számítson. Ezzel más ágra is akár áttehető egy meglévő commit. **VESZÉLYES PARANCS, CSAK ÉSSZEL HASZNÁLANDÓ.**
- **Pull kérelem (pull request, PR):** tegyük fel, hogy egy GitHubos távoli tároló tartalmán változtatásokat hajtottunk végre. Ha van hozzá jogosultságunk, megtehetjük, hogy közvetlenül a tároló fő (main, master) ágába küldjük fel a változtatásainkat, de gyakran megesik, hogy ehhez nincs jogosultságunk. Ilyenkor *pull kérelem* kiadásával kérhetjük meg a távoli tároló üzemeltetőit, hogy változtatásainkat *húzzák be* a főágba. A GitHub felülete lehetőséget nyújt arra, hogy a változtatások hatását megbeszéljük a változtatásainkat ellenőrző (review) üzemeltetőkkel.

## Példa munkafolyamatok

Ebben a részben példa parancssorozatokot tüntetünk fel, melyek különböző, gyakran előforduló helyzetben használhatóak. **Repo létrehozása és remote-hoz kötése**

```
git init
git add <kezdő fájlok>
git commit -m "<Első commit üzenet>"
git branch -M <alapértelmezett ág neve>
git remote add <Remote neve, tetszőleges, tipikusan origin> <Remote elérési útja>
git push -u <remote neve> <alapértelmezett ág neve>
```

## Új ág létrehozása és az ágra váltás, majd ottani munka

```
git checkout -b <új ág neve>
<munka>
git add <módosított fájlok>
git commit -m <Beszédes commit üzenet>
git push --set-upstream <remote neve, tipikusan origin> <új ág neve>
```

## Aktív ágon dolgozás

```
<munka>

Git add <módosított fájlok>
Git commit -m <beszédes commit üzenet>
Git pull
<merge conflictok feloldása>
Git add <merge conflictban módosított fájlok>
Git commit -m <merge-t leíró commit üzenet>
Git push
```

## Munka átváltása másik ágra

```
git stash git checkout  
<cél ág neve> git stash  
pop
```

## Ágazási stratégia

A git legnagyobb ereje a különböző ágak kezelésében rejlik. Nagyobb projektek esetén elengedhetetlen nem csak a commitok tisztán tartása és megfelelő használata, de a különböző fejlesztések elválasztása is. Az ágak ezt teszik lehetővé. Minden ágnak külön szerepe van és csak az oda releváns változtatásokat szabad rájuk commitolni. Ha egy változtatás nem fér be sehova, akkor egy új ágot érdemes rá létrehozni.

## A különböző ágak

### Master/Main/Default

Ezen az ágon mindig stabil állapot szerepel. Ha valaki találkozik a repóval, ezzel az ággal találkozik először és az lesz a feltételezése, hogy ami itt szerepel, az a használható termék. Ezen az ágon nem történik fejlesztés és csak a többi ágon már kitesztelt változtatások kerülhetnek ide. Emiatt tipikusan le van maradva a legfrissebb változattól, akár hónapokkal is.

### Dev/Development

Ezen az ágon az éppen legfrissebb, de működő állapot szerepel. A fejlesztés menetében egy-egy új komponens a saját ágán készül el, majd ebbe az ágba olvasztják be, amint kész. Mikor összegyűlik elég komponens, ez az ág egy állapota olvad bele a fő ágba.

### Feature/komponens ágak

Ezek az ágak tipikusan valamilyen jegykövető rendszer alapján vannak elnevezve. Ennek hiányában leíró, könnyen érthető és egyértelmű neveik vannak. A termék új komponensei, hibajavításai vagy változtatásai ilyen ágakon kerülnek fejlesztésre és innen kerülnek be vagy egy nagyobb komponens ágára, vagy a dev ágra.