

Git verziókezelő

Készítette: Hugyák Tamás

*Pannon Egyetem
Műszaki Informatikai Kar*

2017.02.17. – v1.1

Tartalom

1. Git használata	4
1.1. Bevezetés	4
1.2. Fogalmak	4
1.3. Fájlok állapotai	6
1.4. A repository-k fajtái.....	7
1.5. A Git felépítése és egy rövid ismertető.....	7
1.6. Repository-k közötti kommunikáció	8
2. A Git-ről részletesebben.....	11
2.1. Branch-ek	11
2.2. Konfliktus.....	12
2.3. Remote és local branch-ek.....	13
2.4. Saját adatok beállítása	13
2.5. .gitignore fájl	13
3. Git parancsok ismertetése	15
3.1.1. clone	15
3.1.2. status.....	15
3.1.3. add.....	15
3.1.4. commit	15
3.1.5. checkout.....	16
3.1.6. fetch	16
3.1.7. push.....	16
3.1.8. pull.....	17
3.1.9. revert.....	18
3.1.10. merge	18
3.1.11. branch	18

3.1.12. diff	19
3.1.13. reset	19
3.1.14. tag.....	19
3.1.15. stash	20
3.1.16. log.....	20
3.1.17. rm	20
3.1.18. mv.....	21
3.2. Néhány ábra a Git parancsokról.....	21
3.3. checkout vs reset	22

1. Git használata

1.1. Bevezetés

A Git egy nyílt forráskódú, elosztott verziókezelő szoftver, mely a sebességre helyezi a hangsúlyt. A fejlesztők a saját gépükön nem csak a repository-ban (tárolóban) lévő legfrissebb állapotát tárolják, hanem az egész repot.

A verziókezelői tevékenységek végrehajtása nagyon gyorsan történik, mely a Git erősségét is adja. A központi szerverrel történő hálózati kommunikáció helyett a lokális, saját számítógépen hajtódnak végre a parancsok, így a fejlesztés offline megy végbe a workflow megváltoztatása nélkül. A központi repository-val csakis akkor történik kommunikáció, hogyha arra a felhasználó parancsot ad.

Mivel minden egyes fejlesztő lényegében teljes másolattal rendelkezik az egész projektről, ezért a szerver meghibásodásának, a tároló megsérülésének vagy bármilyen bekövetkező adatvesztésnek a kockázata sokkal kisebb, mint a központosított rendszerek által támasztott pont-hozzáférés esetében, hiszen bármely lokális repository-ból visszaállítható a központi szerverre az eredeti tároló állapota.

A Git repository minden egyes példánya – akár local, akár remote – rendelkezik a projekt teljes verziótörténetével, így egy elszigetelt fejlesztői környezetet biztosít minden fejlesztő számára, hogy szabadon kísérletezzenek új funkciók fejlesztésével mindaddig, amíg egy tiszta, publikálható verziót nem képesek előállítani.

1.2. Fogalmak

A Git hasonló egy hash-fához, azonban az egyes csomópontokon és leveleken hozzáadott adatokkal rendelkezik.

A Git célja az adott projekt menedzselése, ill. az adatok változásának nyomon követése. Ezen információk adatstruktúrákban történő tárolását *repository*-nak, röviden *repo*-nak, avagy lokális adatbázisnak nevezik.

A *working directory*, *working copy* vagy *history* az adott projektről, a gyökérkönyvtárról – amelyben a fájlok, forráskódok és mappák találhatóak – egy változatot, verziót, állapotot tartalmaz.

Snapshot egy adott pillanatban, időpontban a könyvtárak, fájlok aktuális állapotát, tartalmát, verzióját jelenti. A pillanatkép tulajdonképpen nem más a fájlok esetében, mint egy teljes másolat azok tartalmáról.

Commit-olásnak nevezik azt a folyamatot, amely során a Git a *megjelölt* és a *staging area*-ben lévő fájlokról készült snapshot-okat a lokális adatbázisában (a *.git* könyvtárban) eltárolja, és a tartalmuk alapján egy *SHA-1 hash* kódot (*commit id*) generálva hivatkozik rájuk.

A Git lehetővé teszi, hogy a módosult fájlok közül egy csoportban csak azok kerüljenek eltárolásra az adatbázisában, amelyeket a fejlesztők kiválasztottak. Ezért a *working directory* és a lokális adatbázis közé egy harmadik, *index*, *cache*, *staging area* szinonima nevekkkel illetett átmeneti területet alakítottak ki, amelyben információk szerepelnek arról, hogy a következő *commit*-ban mely snapshot-ok legyenek eltárolva. Röviden a *staging area* a fájlrendszernek a következő *commit*-ra jelölt elemei snapshot-jait tartalmazza.

Staged, *cached* jelzővel illetik azokat a fájlokat, amelyek verziókezelve vannak a Git által és a legutolsó *commit* óta módosítva lettek, illetve az állapotukról, verziójukról, tartalmukról *már* készült snapshot.

A *working directory*-t *tisztának* (*clear*) nevezik, ha a fájlokon végzett *összes* módosítás el van mentve a lokális Git adatbázisba, a *repo*-ba. Ilyenkor az utolsó *commit* óta nem történt változtatás az adatokon. A *working directory*-t *piszkosnak* (*dirty*) nevezik, ha a legutolsó *commit* óta a verziókezelt fájlokon történtek olyan változtatások, módosítások, amelyek még nem lettek *stage*-elve, azaz a *staging area*-ba helyezve.

Branch-nek nevezik azon *commit*-ok összességét, melyek egy közös ágra lettek rendezve, és egy közös ősi *commit*-ból erednek lineárisan egymás után fűzve és hivatkozva.

HEAD az aktuális lokális *branch*-ben a legutolsó *commit*-ra való hivatkozás. Ha a *HEAD* által mutatott *commit*-tól kezdve az egyes *commit*-ok szülőjén felfelé lépünk,

akkor egy elágazásmentes úton haladva a repository legelső commit-jához, a gyökérhez jutunk. Ezt az utat nevezzük branch-nek.

Egy HEAD-et *detached* tulajdonsággal illetik, ha az aktuális commit, amelyre mutat, egyik lokális branch-nek sem a *legutolsó* commit-ja.

A kiadott Git parancsok minden esetben az aktuális branch-re vonatkoznak. Mindig létezik egy aktív, kiválasztott aktuális lokális branch.

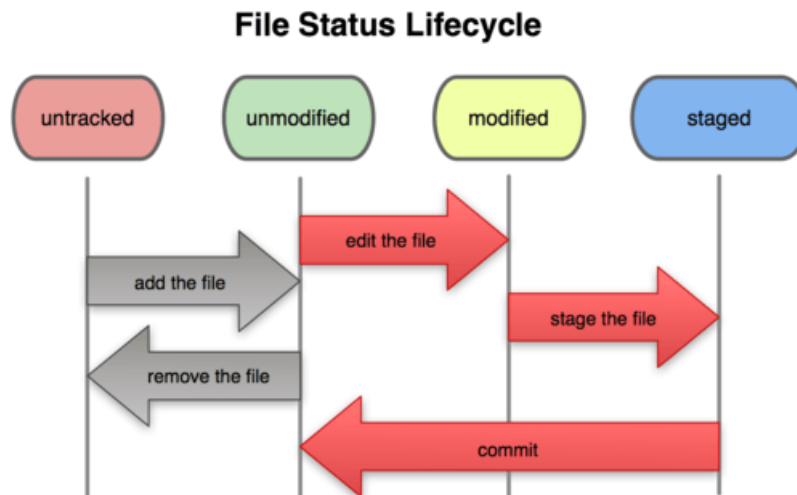
1.3. Fájlok állapotai

A Git *untracked* jelzővel illeti azokat a fájlokat és mappákat, amelyekről még egyetlen snapshot sem készült (nem szerepelnek a staging area-ban, ill. nem része a lokális adatbázisnak), tehát nem követi nyomon a rajtuk végzett módosításokat, azaz nincsenek verziókezelve.

Tracked megjelöléssel szerepelnek azok az állományok és könyvtárak, amelyek tartalmának változását a Git nyomon követi, verziókezeli.

A *tracked* jelöléssel rendelkező fájloknak 3 további állapotuk lehetséges:

- *modified*: a fájl (tartalma vagy neve, kiterjesztése) módosult a legutóbbi commit óta, de az aktuális állapota még nincs a staging area-ban (nem készült róla snapshot, nincs stage-elve), ezáltal a következő commit-nak nem lesz része.
- *unmodified*: a fájl nem módosult a legutolsó commit óta.
- *staged, cached*: a fájl módosult a legutóbbi commit óta és már snapshot is készült róla (a staging area része, stage-elve lett), a következő commit során el lesz tárolva az adatbázisban.



1. ábra: A fájlok lehetséges állapotai.

1.4. A repository-k fajtái

Egy adott repo-nak két fajtáját különböztetik meg elhelyezkedés szerint:

- *remote*: a távoli, szerveren lévő központi repo-t jelenti, mely segítségével tartják a kapcsolatot egymással a fejlesztők repo-jai; alapértelmezetten az *origin* névvel hivatkoznak rá
- *local*: a helyi számítógépen lévő, klónozott repo; a tényleges fejlesztés ezen történik

Egy adott repo-nak két fajtáját különböztetik meg hozzáférhetőség szerint:

- *private*: a repository-hoz csak regisztrált és engedélyezett felhasználók férhetnek hozzá
- *public*: a repository-hoz bárki hozzáférhet és használhatja

1.5. A Git felépítése és egy rövid ismertető

A Git 3 alapkövét és részét a working directory, a staging area és a local database (.git könyvtár) jelenti.

Working Copy

Your Project's Files



Git watches tracked files for new local modifications...

Staging Area

Changes included in the Next Commit

Local Repository

The ".git" Folder

Tracked (and modified)



If a file was modified since it was last committed, you can stage & commit these changes



Changes that were added to the Staging Area will be included in the next commit



All changes contained in a commit are saved in the local repository as a new revision



Changes that are **not staged** will not be committed & remain as local changes until you stage & commit or discard them

Untracked



If a file was modified since it was last committed, you can stage & commit these changes

2. ábra: A Git felépítése.

Egy új, üres repo létrehozása után a gyökerkönyvtár `.git` nevezetű, rejtett mappájában helyezi el a Git verziókezeléshez szükséges fájljait. Ezután a gyökerkönyvtárban bármilyen adatot elhelyezve a Git érzékelné fogja a változást. Például egy új, üres fájl létrehozásakor a verziókezelő jelzi, hogy egy olyan fájlt talált, amely még nem része az index-nek (pontosabban nincs verziókezelve), ezáltal úgynevezett *untracked* minősítéssel illeti. Ebben az esetben a rendszer nem követi nyomon a fájlon végrehajtott módosításokat, csak jelzi, hogy nincs indexelve. Ahhoz, hogy a rendszer figyelje a fájlt, s a rajta végzett változtatásokat, hozzá kell adni az index-hez (stage vagy add parancs). Amikor ez megtörténik, *tracked* minősítést kap, és egy pillanatkép (másolat) készül róla, mely az aktuális tartalmát jelöli. Ezután egy üres könyvtár létrehozását is *untracked*-ként fogja megjeleníteni, melyet a cache-hez történő csatolás után *tracked*-re módosít a Git. Az első commit-olás után a fájl és könyvtár aktuális állapota (tartalma) mentésre kerül az adatbázisba és *unmodified* minősítést kapnak.

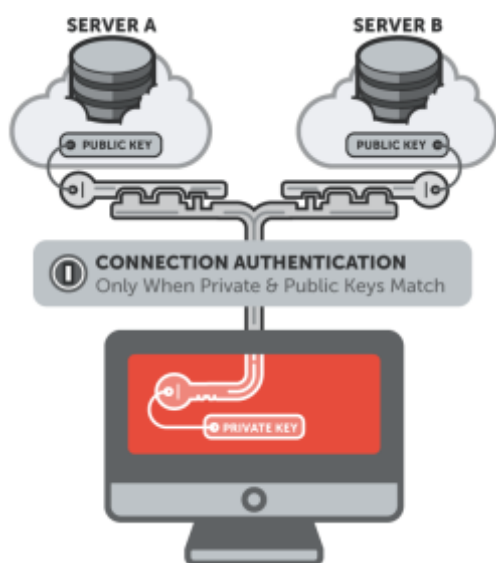
1.6. Repository-k közötti kommunikáció

A Git-es parancsok használata során – néhányat kivéve – a verziókezelő rendszer nem hoz létre (interneten keresztül kommunikációs) kapcsolatot a *local* (saját gép) és *remote*

(központi szerver) repo között. Kommunikációra csak akkor van szükség, amikor az adatbázisok szinkronizációja történik. Ez történhet a HTTPS protokollal vagy az SSH nyilvános kulcsai segítségével. Mindkét esetben titkosított csatornán folyik a kommunikáció.

A HTTPS protokollt használva a szolgáltató weboldalán (például Github, Bitbucket, Gitlab) regisztrálni kell egy felhasználónév - jelszó párossal, amelyre a csatlakozás során lesz szükség. (Természetesen ezután egy repository-t szükséges létrehozni a szolgáltató weboldalán.) Ezt követően a repo klónozása alkalmával a szerverhez történő kapcsolódás során a Git kérni fogja a felhasználónevet és a hozzá tartozó jelszót (amelyekkel a szolgáltatónál a regisztráció történt meg), s a sikeres autentikáció után megtörténik a tároló letöltése. Repo klónozása HTTPS protokoll segítségével az alábbi parancs kiadásával valósítható meg a terminálban:

```
git clone https://szolgáltato/felhasznalonev/repo-neve.git
```



3. ábra: Kapcsolódás Git szolgáltatóhoz az SSH protokoll segítségével és a privát-publikus kulcspáros használatával.

SSH protokollt használva először a sajátgépen egy privát-publikus RSA kulcspárost szükséges generálni, mely a kommunikáció alapját szolgálja. Ez a művelet az alábbi parancs kiadásával hajtható végre, amely 4096 bites RSA kulcspárost fog létrehozni:

```
ssh-keygen -t rsa -b 4096
```

A generálás során egy *passphrase*-t kell megadni, amely jelszóként szolgál majd a Git kommunikáció során. A generált kulcsok az operációs rendszer aktuális felhasználója főkönyvtárának a *.ssh* nevezetű rejtett mappájában találhatóak. A publikus kulcs *.pub* végződésű, míg a privát kulcs nem rendelkezik kiterjesztéssel, s alapértelmezetten mindkettő fájlnak *id_rsa* a neve. A privát kulcs nem kerülhet másik fél kezébe, ügyelni kell annak biztonságára. Az *id_rsa.pub* publikus kulcs tartalmát az adott Git szolgáltató oldalán történő bejelentkezés után a *Beállítások*

menüpont alatt az *SSH kulcsok*-at kiválasztva fel kell venni saját kulcsként. (Mindez azért szükséges, hogy a Git kommunikáció során – amely a saját gépemen a privát kulccsal történik meg – a szolgáltató a feltöltött publikus kulcs alapján tudja, hogy ahhoz én rendelkezem hozzáféréssel.) Ezután már az SSH protokoll használatával klónozható is a saját repository-m a szolgáltatótól a következő parancs kiadásával:

```
git clone git@szolgáltato:felhasznalonev/repo-neve.git
```

A csatlakozás során a rendszer kérni fogja a passphrase-t, melyet a kulcsok generálása során kellett megadni. A helyes jelszó begépelése után a Git letölti a tárolót a remote szerverről az aktuális mappába úgy, hogy az a *repo-neve* nevet kapja. A szolgáltatónál létrehozott összes repository klónozható a feltöltött publikus kulcs segítségével, tehát nincs szükség minden egyes repo-hoz külön kulcspárost létrehozni.

2. A Git-ről részletesebben

Jelen fejezetek a Git részletesebb információkat közölnek, amelyek nélkülözhetetlenek a használatának elsajátításához.

2.1. Branch-ek

A *branch* a fejlesztés egy ágát jelenti, melyet névvel szoktak ellátni. A branch-ek lehetővé teszik, hogy a fejlesztés ne csak egy szálon történjen. Az egyidejű, párhuzamos fejlesztések egymástól elszeparálva működnek. Egy új repo létrehozásakor automatikusan létrejön egy alapértelmezett branch: a *master*.

A projekt history commit-okból épül fel, melyek egymásra hivatkoznak. Minden egyes commit-nak legalább egy, legfeljebb kettő szülője van (merge-ölés után). Branch létrehozása során ki kell jelölni egy bázis commit-ot, melyből az új branch származtatása történik. Amint olvasható, a bázis commit-ig mindkét branch (az új és a régi, amelyben a bázis commit található) rendelkezik (pontosabban osztozik) ugyanazon commit-okkal. A branch-ek egymással párhuzamosan léteznek.

Megjegyzés: A branch-ek egyszerű módon úgy képzelhetőek el, mintha a repo-t (tehát a projekt gyökérványtárát) egymás mellé többszörösen lemásolnánk (CTRL + C, majd CTRL + V), s az egyes másolat könyvtárakat a branch nevekkkel illetnénk.

A branch tulajdonképpen arra szolgál, hogy a fejlesztést különböző részekre (pl. fejlesztési fázisokra és modulokra) lehessen osztani. Ez azért fontos, mert a fejlesztők egymástól függetlenül dolgozhatnak az egyes ágakon ugyanazon repo-n. Lehetőség van branch-ek közötti váltásra és branch-ek egybefűzésére is. Az előbbi azért fontos, mert átjárhatóságot biztosít a fejlesztés különböző részei között, az utóbbi pedig a fejlesztés párhuzamosan folyó meneteit képes egybeolvasztani, ezáltal kettő vagy több eredményből egy közös keletkezik.

Az egész Git repository-nak a gyökerét az első commit jelenti. Az összes commit és branch ebből a commit-ból, vagy ennek leszármazottjaiból ered.

Minden tároló rendelkezik legalább 1 branch-csel. A fejlesztés az ágakban történik. Branch-ek közötti váltás során a Git visszakeresi a két branch legutolsó közös commit

hivatkozását (bázis vagy ős branch), s attól a commit-tól kezdve a cél branch összes módosítását leírja a fájlrendszerre.



4. ábra: Branch-ek kialakítása.

Az ágak létrehozásakor a *master* branch legtöbbször a *production* fázist jelöli, s ebből származtatják a *release*, *develop* és *hotfixes* ágazatokat. A fő fejlesztés a *develop* ágon megy végbe, melyből modulonként 1-1 *feature* branch-et származtatnak. Az egyes modulok elkészülte után a *develop* ágba fűzik a kész *feature* branch-eket. A *release* branch-be a tesztelés alatt álló *develop* verziók kerülnek. Ha az adott *release* verzió stabil, akkor a végleges *master* ágba kerül befűzésre. A *master* branch-ben felmerült hibákat a *hotfix* nevezetű branch-ekkel javítják, melyeket közvetlenül a *master*-ből származtatnak.

2.2. Konfliktus

Konfliktus történik branch-ek merge-ölése vagy pull-olás során, ha két különböző commit egy fájl ugyanazon során történt változtatást tárolja. Ez esetben a Git nem tudja eldönteni, hogy mely módosításokat hagyja meg vagy törölje, ezért értesíti a fejlesztőt, hogy manuálisan javítsa ki a konfliktust.

Például konfliktus történik, ha 2 fejlesztő a *master* branch-en dolgozik, s egymás után commit-olnak úgy, hogy egy fájl egy adott sorát mind a ketten szerkesztették. Ez esetben az időben később push-oló fejlesztő kap egy figyelmeztetést (*rejected*), hogy a remote

szerveren találhatóak olyan commit-ok az aktuális branch-ben, amelyek még nem lettek letöltve a lokális repoba, ezért még nem push-olhatóak a lokális commit-ok. A fejlesztőnek ez esetben le kell töltenie a változtatásokat (*pull*), s ekkor értesül majd a konfliktusról, hogy a fájl egy sorát már más is módosította. Az adott fájlt szerkeszteni kell, össze kell vágni a helyes kódsort a két commit-ból. Ezután újra kell commit-olni a változtatásokat, s majd már a feltöltés (*push*) művelet sikeresen végrehajtható.

2.3. Remote és local branch-ek

Egy klónozott, sajátgépen lévő repo-ban *remote* és *local* branch-ek találhatóak. A *remote* branch-ek referenciák a *remote* repo-ban lévő branch-ekre. Amolyan „könyvjelzőként” és emlékeztetőül szolgálnak, hogy a *remote* repo branch-eiben milyen commit-ok szerepelnek.

A *remote* repository-kban csak a *remote* branch-ek léteznek, *local* branch-ek nem.

A *local* branch-ek a fejlesztést teszik lehetővé, s a commit-ok feltöltése a *remote* tárolóba a *local* branch-ekből történik.

A *remote* branch-ek a *remote* repository-nak (összes branch-ének) az állapotát tárolják, míg a *local* branch-ek a sajátgépen történő fejlesztést tartalmazzák.

2.4. Saját adatok beállítása

A Git commit-ok létrehozásakor eltárolja a parancsot kiadó felhasználó nevét, e-mail címét, dátumot és egyéb információkat. A következő parancsokkal a saját név és e-mail cím állíthatóak be:

```
git config --global user.name "Név"  
git config --global user.email "e-mail"
```

2.5. .gitignore fájl

A *.gitignore* fájl tartalmazza azon, a repository-ban lévő fájlok és könyvek nevét, amelyeket a Git nem követ nyomon, nem figyeli a változásait, kihagyja a verzió nyomon követésből. A fájlok és mappák neveiben reguláris kifejezések is szerepelhetnek. Általában a projekt gyökérkönyvtárában helyezik el. Példa a tartalmára:

```
# Debug és debug mappák kihagyása
```

```
[Dd]ebug/  
# az összes .txt kiterjesztésű fájl kihagyása  
*.txt  
# obj könyvtár kihagyása  
obj/  
# config.c fájl kihagyása  
config.c  
# lib/main.c fájlt kövesse, nem lesz a kihagyott fájlok között  
!lib/main.c
```

3. Git parancsok ismertetése

A következő fejezetekben a Git parancsainak ismertetése történik.

3.1.1. clone

A *clone* egy olyan parancs, mely segítségével egy repository lemásolható a sajátgépre.

```
git clone git@host:felhasznalonev/repo-neve.git repo-mas-neve
```

A Git az aktuális könyvtárban létrehoz egy *repo-mas-neve* nevezetű mappát, majd ebbe a könyvtárba tölti le a tároló adatait, a teljes repo-t. A *repo-mas-neve* paraméter elhagyható, mely esetén a repo nevével megegyező, *.git* kiterjesztés nélkül hozza létre a könyvtárat az aktuális mappában.

3.1.2. status

Az index tartalmának és a working directory állapotának megjelenítésére szolgál. Az *untracked* fájlok, ill. a *modified* és *staged* adatok listázására ad lehetőséget.

```
git status
```

3.1.3. add

Az *add* parancs a módosult adatokat az index-be helyezi, snapshot-ot készít róluk. Az összes fájl cache-be történő helyezése rekurzívan:

```
git add .
```

Csak a *config.c* fájl helyezése a cache-be:

```
git add config.c
```

3.1.4. commit

A módosítások, snapshot-ok eltárolására szolgál a lokális adatbázisban. Minden commit rendelkezik egy rövid szöveggel, leírással, mely arra utal, hogy milyen módosításokat tartalmaz. A commit parancs futtatása csakis az *add* parancs kiadása után lehetséges!

```
git add .  
git commit -m "a commit szövege"
```

Az *add* parancs elhagyható az *-a* kapcsolóval csakis akkor, ha az összes módosult fájl *tracked* minősítésű. Egyéb esetben a Git hibát ad:

```
git commit -am "a commit szövege"
```

A *-m* kapcsoló elhagyásával az alapértelmezett szövegszerkesztő ugrik fel, s abban írható meg a commit szövege:

```
git commit
```

3.1.5. *checkout*

A *checkout* parancs a branch-ek közötti váltásra szolgál. Ilyenkor a Git az aktuális branch commit-jain visszafelé haladva megkeresi a legelső, cél branch-csel ugyanarra hivatkozó commit-ot, s attól fogva a cél branch commit-jainak változtatásait leírja a fájlrendszerre. A következő példában a *develop* branch-ra történik váltás:

```
git checkout develop
```

3.1.6. *fetch*

A *fetch* parancs segítségével a remote repo-n lévő commit-ok importálásra kerülnek a lokális adatbázisba. A parancs csak a lokális adatbázist frissíti, a *working directory*-t és a *staging area*-t érintetlenül hagyja. A letöltött commit-ok a *remote* branch-ekben tárolódnak a *local* branch-ek helyett, ezáltal lehetőség van a szerveren történt módosításokat áttekinteni, mielőtt azok integrálva (*merge*) lesznek a lokális branch-be.

```
git fetch <remote_repo> [<branch_neve>]
```

Például:

```
git fetch origin
git fetch origin master
```

3.1.7. *push*

Az aktuális branch commit-jainak (snapshot-jainak) feltöltése a remote repository egy meghatározott branch-ébe. *Remote* repository-nak nevezzük a távoli szerveren lévő tárolót, mely segítségével tartják egymással a kapcsolatot a fejlesztők. A *remote* szerverre alapértelmezetten *origin* névvel hivatkoznak.

A *fetch* parancs párjaként is emlegetik, mert a *fetch* importálja, a *push* pedig exportálja a commit-okat a *local* és *remote* repository-k között.

Használata:

```
git push -u <remote_repo> <branch_neve>
```

Például: az aktuális branch feltöltése a remote szerver *master* branch-ébe:

```
git push -u origin master
```

A branch nevét azért szükséges megadni, mert előfordulhat, hogy jelenleg a *master* branch az aktuális ág, míg a módosításokat a távoli szerver *production* branch-ébe szeretnénk feltölteni, nem a *master* branch-ébe:

```
git checkout master           # master ágba lépés
git push -u origin production # snapshot-ok feltöltése nem a master
                              # ágba, hanem a production-be
```

3.1.8. pull

A központi szerverre felküldött változtatások, commit-ok letöltése és merge-ölése az aktuális branch-be. Az adatbázis frissítése (*fetch*) után a *working directory*-ba, a fájlrendszerre is leírja (*merge*) a módosításokat.

```
git pull [<remote_repo> <branch_neve>]
```

A *<remote_repo>* és *<branch_neve>* paraméterek opcionálisak. Például:

```
git pull
git pull origin
git pull origin master
```

A *git pull origin master* parancs a következőt jelenti:

```
git fetch origin
git merge origin/master
```

A *pull* parancs *rebase* móddal is működtethető. Ez esetben az aktuális branch lineáris marad. Az adatbázis frissítése (*fetch*) után a *working directory*-ba, a fájlrendszerre is leírja (*rebase*) a módosításokat:

```
git pull --rebase [<repository> <branch_neve>]
```

3.1.9. *revert*

A *revert* parancs segítségével egy commit-tált snapshot összes módosítása visszavonható. A parancs esetében a Git nem törli a korábbi commit-ot, hanem az aktuális branch-ben létrehoz egy újat, amely visszavonja a megadott commit változtatásait. Erre az integritás megőrzése miatt van szükség.

```
git revert <commit_sha-1_azonosító>
```

3.1.10. *merge*

A *merge* parancs lehetővé teszi önálló fejlesztési ágak integrálását egy ágazatba. A parancs használata során annak a branch-nek a nevét kell megadni, amelyet a jelenleg aktív ágba szükséges integrálni. Tehát a *merge* paranccsal másik ág integrálása történik az aktuális ágba.

```
git merge <branch_neve>
```

3.1.11. *branch*

A *branch* parancs a branch-ek listázására, létrehozására és törlésére szolgál.

A következő parancs egy új ágat hoz létre úgy, hogy a bázisát majd az a commit képezi, amely az aktuális branch aktuális commit-ja:

```
git branch <branch_neve>
```

Branch-ek listázása a következő módokon tehető meg:

```
git branch  
git branch --list --all
```

Branch törlése kétféle módon lehetséges: a *-d* kapcsolóval biztonságosan törölhető, mert a Git jelez, ha olyan commit-okat tartalmaz, amelyek egyetlen másik ágba sem lettek merge-ölve.

```
git branch -d <branch_neve>
```

A *-D* kapcsolóval oly módon törölhető egy branch, hogy lehetnek benne olyan commit-ok, amelyek egyetlen egy másik ágazatnak sem része.

```
git branch -D <branch_neve>
```

Az aktuális branch átnevezése:

```
git branch -m <uj_branch_nev>
```

3.1.12. *diff*

A *diff* parancs a változtatásokat mutatja meg a working directory és az index, vagy két commit, branch, esetleg fájl között.

```
git diff
git diff --cached
git diff <branch1> <branch2>
```

3.1.13. *reset*

A *reset* parancs használható commit-tált snapshot-ok törlésére, ill. a staging area és az index változtatásainak visszavonására. Mindkét esetben csak lokális változtatások visszaállítására alkalmas. A *remote* repo-ra kiküldött snapshot-okra nem alkalmazható!

```
git reset <commit_sha-1_azonosito>
```

3.1.14. *tag*

A Git lehetőséget ad a history fontosabb pontjainak megcímkezésére. Ezt a leggyakrabban a verziószámok megjelölésére használják.

A következő parancs 'v1.4'-es címkével látja el az aktuális commit-ot, és 'my version 1.4' rövid leírással illeti azt.

```
git tag -a v1.4 -m 'my version 1.4'
```

Címkék listázása:

```
git tag -l
```

Adott címke részletes információi:

```
git show <tag_neve>
git show v1.4
```

Csak a címkék felküldése a *remote* repo-ra:

```
git push <remote> --tags
```

3.1.15. *stash*

A *stash* parancs biztosítja a working directory változtatásainak ideiglenes elmentését egy biztonságos helyre. Erre akkor lehet szükség, amikor pl. branch-váltás történik. Egyik ágról a másikra csak úgy lehet átváltani, hogy ha a working directory „tisztá”, tehát vagy nem történtek változtatások, vagy a változtatások már commit-elve lettek. A másik branch-re történő átállás előtt először a módosításokat stash-elni kell – ha a fejlesztő nem szándékozik még commit-olni –, majd ezután engedélyezett a *checkout* parancs futtatása. Stash-elés nélkül a Git hibát ad a folyamatra.

A még nem snapshot-olt módosítások ideiglenes eltárolása:

```
git stash
```

A még nem snapshot-olt módosítások ideiglenes eltárolása 'message' néven:

```
git stash save "message"
```

Ideiglenesen eltárolt változtatások listázása:

```
git stash list
```

A *stash@{1}* módosítások visszaállítása a working directory-ba:

```
git stash apply stash@{1}
```

A legutolsó mentett stash állapot visszaállítása:

```
git stash pop
```

3.1.16. *log*

A commit logokról és a branch-ekről ad információt.

```
git log
```

```
git log --oneline --graph --decorate
```

```
git log --graph --parents --name-status --oneline
```

3.1.17. *rm*

Fájlok törlésére szolgál a working directory-ból és az index-ből. A következő parancs kiadása után mind a fájlrendszerrel, mint az index-ből eltávolításra kerül a fájl:

```
git rm <fajl_nev>
```

Például:

```
git rm readme.txt
```

A `--cached` kapcsolóval csak az index-ből törlődik a fájl, a fájlrendszerrel (working directory) viszont nem:

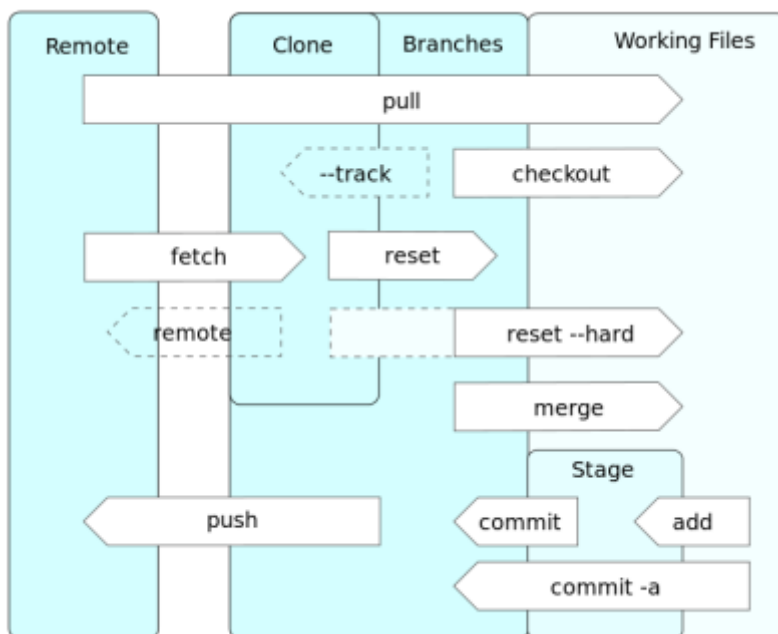
```
git rm --cached readme.tx
```

3.1.18. mv

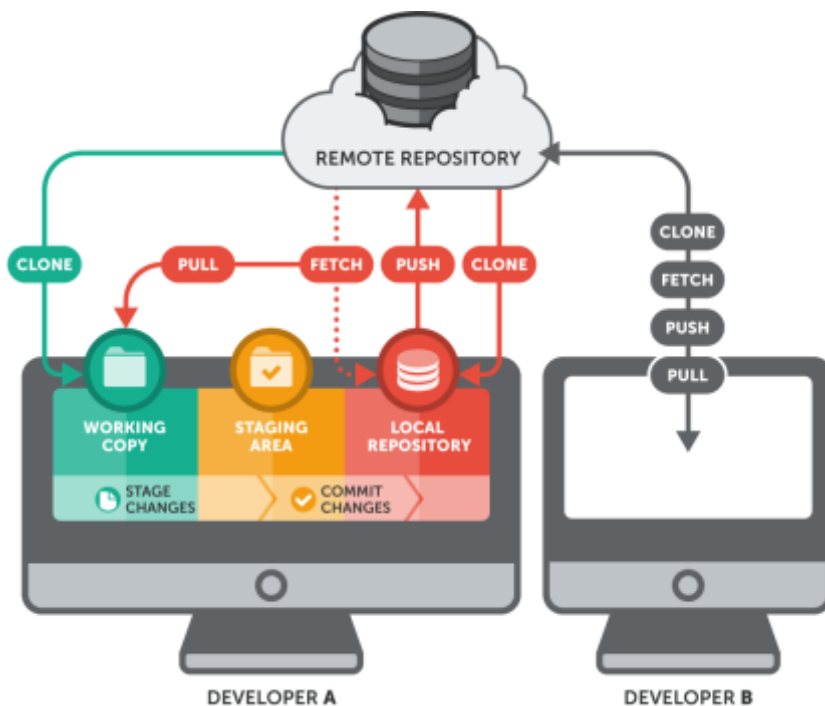
Fájl, könyvtár átnevezésére vagy áthelyezésére szolgál.

```
git mv <forras> <cel>
```

3.2. Néhány ábra a Git parancsokról



5. ábra: Parancsok és szekciók.



6. ábra: A Git workflow.

3.3. checkout vs reset

A *checkout* és *reset* parancsok segítségével korábbi commit-ok állapotára lehetséges visszaállítani a working directory-t. A két parancs használata abban különbözik egymástól, hogy *checkout* alkalmazása esetén az adott commit-ra egy új, ideiglenes branch jön létre (nem változtatja meg az aktuális branch-et), melynek a nevét a commit SHA-1 azonosítója adja. *Reset* használata során az aktuális branch *HEAD*-jét állítja át a Git, ezáltal a branch commit-jai módosulnak.