GIT #2

olvasnivaló:
https://git-scm.com/book/en/v2

Miért kell a verziókezelés?

Munkánk során a forráskód (és dokumentumok is) folyamatosan változnak, viszont néha elfordul, hogy vissza kell tudnunk keresni egy korábbi vagy egy alternatív verziót. Vagy azért, mert valami nem megy és meg akarjuk keresni, hogy hol romlott el (“Tegnap még ment, csak azóta javítottam valamit…”), vagy mert eleve több verziót kell karbantartanunk (használt verzió, teszt verzió, fejleszti verzió stb.).

A “most jó, ezt most így be-ZIP-elem” megközelítés egy ideig mködik, de ha több fejleszt is van, nagyon hamar káoszhoz vezet. Ráadásul a forráskód egymás közti megosztása sem triviális: szerveren lév ZIP, forráskód egy Dropbox megosztásban, fejleszt monogramjával kezdd és dátumot is tartalmazó ZIP fájlnév… ezeknek nem lesz jó vége.

Egy verzió kezel rendszertl elssorban az alábbiakat várjuk el:

- Mindig lehessen tudni, melyik az aktuális verzió. És ha többféle aktuális verzió is van, akkor mindegyiket lehessen követni.
- Több fejleszt is dolgozhasson egy projekten. Azt, hogy többen egyszerre módosítanak valamit, vagy meg kell akadályozni, vagy kezelni kell az esetleges ütközéseket.
- Vissza lehessen állítani egy korábbi verziót.

Ezen kívül még elvárhatjuk az alábbiakat is:

- Lehessen offline is dolgozni, vagyis amikor nincsen kapcsolat a verziókezel szerverrel.
- Ideiglenesen félre lehessen rakni változásokat. Vagy azért, mert félkész és még nem alkalmas arra, hogy elmentsük egy új verzióként, vagy egy másik fejlesztnek akarunk átadni bizonyos módosításokat.
- A verziókezel nem utolsó sorban a forráskód biztonságos tárolását is gyakran ellátja. A nagy megbízhatóságú tárolás ugyan nem a verziókelés feladata, viszont az esetek nagy részében mégis a központi verziókezel szerver az, amirl biztonsági mentések is készülnek. (Van bven olyan cég, ahol akár egész lezárt projekteket dokumentációval együtt egy központi SVN szerverre töltenek fel. Egyrészt így biztos helyen van, másrészt könny lekezelni, ha valamiért jön egy “még véglegesebb verzió”.)

Alapvet fogalmak a verziókezelésben

A verziókezelk világában gyakori fogalmak az alábbiak:

- repository: az a tároló, amibl minden korábbi verzió kinyerhet. A változásokat ide mentjük le. Ez valójában lehet egy háttérben lév adatbázis (pl. TFS esetén), egy sima könyvtár a fájlrendszerben (pl. GIT, SVN).
- working directory, munkakönyvtár: ebben a könyvtárban van az aktuális verzió, amin dolgozunk. Errl tudunk egyfajta állapotmentéseket készíteni a repositoryba. Ha vissza kell térnünk egy korábbi verzióhoz, a munkakönyvtár tartalma fogja felvenni a kiválasztott korábbi állapotot.
- commit, changeset: egy változás csomag, ami két egymás utáni verzió különbségét tartalmazza. A verziókezelk általában csak a változásokat mentik el helytakarékossági okokból. Ezek a commitok a projekt elre haladtával hosszú láncokat, vagy egyes esetekben ennél bonyolultabb (irányított) gráfokat alkotnak, ahogy egymásra épülnek, szétválnak több irányba, néha pedig újra összeolvadnak.
- check in, commit, mint mvelet: szintén commitnak hívjuk azt a mveletet, amikor a munkakönyvtár egy állapotát elmentjük, “commitolunk”. Ekkor a repositoryban létrejön egy új commit és az lesz az új aktuális verzió.
- revert: az a mvelet, amikor visszavonunk egy commitot, vagyis visszatérünk egy korábbi verzióra. Például azért, mert negyed óra múlva demózni kell és éppen semmi nem mködik.
- branch: A commitok gráfjának egy ága, aminek van egy aktuális verziója, valamint annak sei. Sok esetben egyetlen egy branch van, de elfordul, hogy például megkülönböztetünk egy fejleszti és egy éles branchet, mivel a fejlesztk nem azon a verzión dolgoznak, ami éles környezetben, éppen használatban van. A projekteknek komoly branchelési stratégiájuk szokott lenni, mely leírja, hogy milyen céllal tartanak fenn külön brancheket.
- trunk: több rendszerben van egy “f branch”, melyet trunknak neveznek. E mögött az a koncepció, hogy itt folyik a f fejlesztés, a többi branch csak ideiglenes, rövidebb távra készült leágazás, ami aztán vagy megsznik, vagy visszaolvad a trunkba.
- merge: ha több fejleszt dolgozik együtt, rendszeresen össze kell olvasztani a változtatásaikat. Ez a merge, vagyis összefésülés mvelete. A verziókezelk nagyon sok esetben ezt automatikusan meg tudják oldani (auto merge). Gond tipikusan akkor van, ha a két fejleszt ugyanannak a fájlnak ugyanazt a sorát módosította, és ezért a merge tool nem tudja eldönteni, hogy mit tegyen. (Nyilván nem lehet az egyiket eldobni, de az sem feltétlenül jó megoldás, ha simán egymás mögé másoljuk a két módosítást.)
- merge conflict: ez az az eset, amikor ütközés van a merge folyamat során. Ezt általában kézzel kell megoldania a fejlesztknek, aminek nagyon nem szoktak örülni. Érdemes gyakran mergelni, mert akkor az egyes fejlesztk munkája nem távolodik el nagyon egymástól.
- 3-way-merge: a mergelési folyamat egyik széles körben alkalmazott módszere. Lényege, hogy a két összefésülend verzió (X és Y) közös sét (A) is megkeresi (amikor még a két ág megegyezett), majd minden ettl való eltérésrl el kell dönteni, hogy az A-beli, X-beli, vagy az Y-beli verzió maradjon meg. Ha a kérdéses sorban X vagy Y közül az egyik megegyezik A-val, akkor a másik az újabb, így azt választjuk. Ha viszont X és Y is változott, akkor van merge conflict, amikor általában a felhasználóra bízzuk a döntést.
- tag: egyes verziókezelk megengedik, hogy bizonyos állapotokat megjelöljünk. Ilyen tag vagy címke lehet például az a verzió a

szakdolgozatból, mely felkerült a diplomaterv portálra, vagy az a firmware verzió, melyet a RobonAUT kvalifikáción használtunk. Nem árt tudni, hogy ha hirtelen kell egy mködképes verzió, akkor melyik volt az.

A Gitrl

A Git egy elosztott verziókövet rendszer. Angolul **Decentralized Version Control System**, röviden **DVCS**, de **Distributed Version Control System** rövidítésének is tartják. Ez azt jelenti, hogy a felhasználó nem csak az aktuális verziót tartja a gépén, hanem az egész tároló (repository) tartalmát, ellentétben a központosított rendszerekkel. Ennek az az elnye, hogy hálózat nélkül is tudunk dolgozni (offline).

Az SVN és más hasonló verziókövet rendszerek központosított verziókövet rendszerek. Angolul **Centralized Version Control Systems**, röviden **CVCS**. A CVCS alapú rendszerek a commitokat a rendszerbekerüléshez viszonyítják. A Git esetén ez nem így mködik. A Git pillanatnyi állapotokat (snapshot) tárol.

A Gitet eredetileg Linus Torvalds fejlesztette ki a Linux kernel fejlesztéséhez. Minden Git munkamásolat egy teljes érték repository teljes verziótörténettel és teljes revíziókövetési lehetséggel, amely nem függ a hálózat elérésétl vagy központi szervertl.

Számos nagy volumen projekt használja jelenleg a Gitet verziókezel rendszerként; a legfontosabbak ezek közül: Linux-rendszermag, GNOME, Samba, X.org, Qt, One Laptop per Child core development, MediaWiki, VLC media player, Wine, Ruby on Rails és az Android platform.[2] A Git karbantartásának felügyeletét jelenleg Junio Hamano látja el.

Elnevezés

A „git” nevet Linus Torvalds némi iróniával a brit angol szleng kellemetlen személyt jelent szavából eredezteti. „Egy egoista szemétláda vagyok, és minden projektemet magam után nevezem el. Elször volt a Linux és most a git.” Ez az önmarcangoló humor valójában csak fikció, hiszen Torvalds valójában küls nyomásra nevezte el a Linuxot maga után. A hivatalos Git wiki számos magyarázatot ad az elnevezésre, mint pl. „Global Information Tracker” (globális információkövet).

Jellemzök

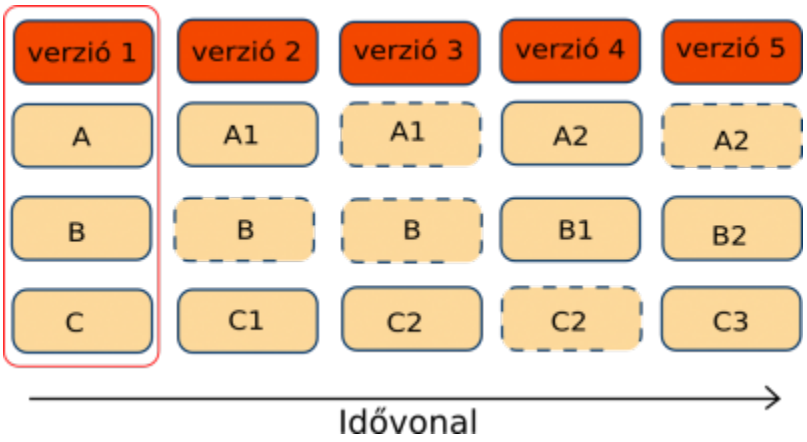
- A nemlineáris fejlesztés ers támogatása. A Git támogatja a branchok készítését, összefésülésüket, és tartalmaz eszközöket a nemlineáris fejlesztési történet ábrázolására. Az alapelgondolás ugyanis az, hogy a Gitben egy változást többször kell összefésülni, mint írni.
- Elosztott fejlesztés. Hasonlóan a Darcs, BitKeeper, Mercurial, SVK, Bazaar és Monotone rendszerekhez, a Git minden fejleszt helyi munkaváltozatában rendelkezésre bocsátja a teljes addigi fejlesztési történetet, és a változtatások másolása mindig két repository között történik. Ezeket a változtatásokat, mint külön ágakat importálják és összefésülhetek, hasonlóan a helyben létrehozott fejlesztési branchokhoz.
- A repositoryk publikálhatóak HTTP-n, FTP-n és rsyncen keresztül, vagy a Git protokollt használva egy socketen vagy SSH-n keresztül is. A Git tartalmaz ezen felül egy Cvs-szeremulációt is, így a már meglév CVS kliensek és IDE pluginok is képessé válnak a Git repository-k elérésére.
- A Subversion és az svk repositoryk közvetlenül használhatóak a git-svn híd segítségével.
- Nagy objektumok hatékony használata. Torvalds a Gitet nagyon gyors és skálázható rendszerként mutatta be. A Mozilla által végzett teljesítménytesztek megmutatták, hogy legalább egy, de bizonyos mveletek esetén akár két nagyságrendnyi sebességfölénye is van minden más verziókövet rendszerhez képest.[4]
- Kriptográfiailag hitelesített történet. A Git-történet olyan módon tárolódik, hogy a szóban forgó revízió neve függ a hozzá vezet fejlesztési történettl. Ha egyszer publikálták, akkor már nem lehetséges észrevétlenül megváltoztatni egy régi revíziót. A struktúra hasonlatos a hash-fához, de hozzáadott adatokkal az egyes csomópontokon és leveleken. (A Mercurial és a Monotone is rendelkezik ezzel a képességgel.)
- Eszközkészlet-szer megoldás. A Git maga C-ben írt programok halmaza, számos shell-szkripttel megtzdelve, amelyek összefzik ezeket a C-ben írt programokat. Annak ellenére, hogy a szkriptek többségét újraírták C-ben a Microsoft Windowsra való portolás eredményeként, a struktúra megmaradt, és továbbra is könny az egyes komponenseket láncba rendezni az igényeknek megfelelen.
- Plugin-rendszer összefésülési stratégiák. Az eszközkészlet-struktúra részeként a Gitnek van egy jól definiált modellje a nem teljes összefésülésre, és számos algoritmusa befejezni azt. Amennyiben ezek az algoritmusok nem alkalmasak a feladatra, úgy a felhasználót értesíti az automatikus folyamat megszakításáról, és kéri a kézi szerkesztést.
- A hulladék adatok felhalmozása, amennyiben nincs összegyjtve. A mveletek megszakítása vagy a változtatások visszavonása használatlan kósza adatokat eredményez az adatbázisban. Ezek általában csak töredékét képezik a folyamatosan növekv mennyiség valós adatnak, de ezen tároló területek felszabadítása a `git-gc --prune` paranccsal meglehetsen lassú tud lenni.

Alkalmas nagy projektek használatára. A Gittel parancssorban dolgozunk, de rendelkezésre áll néhány grafikus felület program is.

Összehasonlítás más verziókövetkkel

A Git rendszer eltér más verziókövet rendszerek mködésétl, mint amilyen a Subversion vagy a Perforce.

A legfbb különbség a Git és más verziókövet rendszerek között az adatok kezelésében van. Más verziókövet rendszerek minden állományról készítenek egy mentést. A Git ezzel szemben, ha egy állomány nem változott, akkor azt nem tárolja el. Csak egy **mutatót** hoz létre az **elz verzióra**.



Vannak rendszerek, amelyek az állományok egy részérl készítenek csak másolatot. A Git csak a változtatott állományról, de az adott állomány egész részérl.

- CVS: egy egyetemi projektben készült az els verzió, amikor egy professzor közösen akart fejleszteni két hallgatójával és eltér volt az idbeosztásuk. 1986, minden verziókezel atyja, máig lehet vele találkozni.
- SVN: A CVS utódja, nagyon elterjedt (de a GIT egyre inkább kiszorítja). Nagyon sok cégnél lehet vele találkozni, ahol centralizált verziókezelt szerettek volna és nem akartak Microsoft technológiára építeni.
- GIT: Eredetileg a Linux kernel fejlesztéséhez alakították ki, ahol nagyon-nagyon sok fejleszt munkáját kellett összefogni.
- Mercurial: letisztult python kód, hasonló a GIT-hez és közel egyszerre indultak útnak. Egyszerbb a gitnél (gyorsabb tanulás) és az elsdleges szempont a teljesítmény volt (nagy projektek számára).
- TFS: A Team Foundation Server a Microsoft technológiai vonalának (centralizált) verziókezelje. Rengeteg integrációs és kollaborációs szolgáltatása van. A TFS jelenleg már támogatja a git repositorykat is, így git szerverként is kiválóan mködik.

Telepítés

Debian alapú Linuxon:

```
apt-get install git
```

Windowsról innen lehet letölteni:

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

Grafikus kliens:

```
apt-get install git-cola
```

vagy:

- http://git-cola.github.io/

vagy ncurses alapú kliens Linuxra:

```
apt-get install tig
```

Szervertelepítés Linuxra az oktatás:linux:git szerver lapon.

Telepítés Windowsra Git Windowsra lapon olvasható.

Commit nézeget:

```
apt install gitk
```

Mködés

Helyi mködés

A gitben, amikor fájlokon és más erforrásokon dolgozunk, minden helyben történik, **nincs** szükség **hálózati kapcsolatra**. Így a munkánk gyors és offline lehet. A munkánkat bármikor szinkronizálhatjuk egy központi tárolóval.

Fogalmak

Kommit

A kommit, a fájlokról elmentett pillanatkép. Érdemes úgy megválasztani a kommitjainkat, hogy az logikailag összetartozó egységeket együtt kommitoljuk. A git rendszerben commit alparanccsal használjuk. A kommit a legelemibb tárolandó egység. Érdemes minél srbben kommitolni.

Branch

Fejlesztés egy ága. Amikor készítünk egy projektet és azt a Git-be helyezzük, lesz egy f (master) baranch-ünk. Ebbl újabb branchokat hozhatunk létre. A háttérben egy branch létrehozása egy mutató az utolsó állapotra. Egy branch tulajdonképpen kommitok összessége.

Clone

Távoli géprl másolatot készíthetünk egy branch-rl.

Checkout

A branchek közötti váltásra használható.

Push

Ha kiadok egy commit parancsot, akkor az csak helyben történik – nem úgy mint az SVN-ben. A központi szerverrel való szinkronizáláshoz szükségünk van a Push mveletre a push paranccsal.

Head

Egy mutató, ami a munkánk során az aktuális branch-ra mutat.

Munkakönyvtár

A munkakönyvtár az a könyvtár, amelyben létrehozzuk a forrásállományainkat, és ahol dolgozunk rajtuk. Ez tulajdonképpen a projekt könyvtára.

Stage

A stage egy átmeneti hely. Azokat az állományokat, amelyeket most hoztunk létre vagy módosítottuk a stage helyre kell tenni, ha szeretnénk követett állapotba tenni.

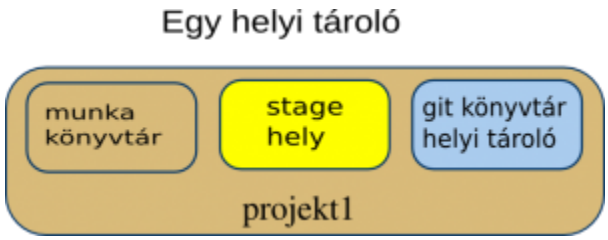
Repository

Tároló, ahol a fájlok tárolásra kerülnek. A helyi gépen a tároló valójában a .git könyvtárban van. A távolig szerveren, egy .git kiterjesztés könyvtár

a szokásos tárolási hely.

Helyi tároló

A helyi tároló a munkakönyvtár, a stage hely és a repository helyek együttese.



upstream

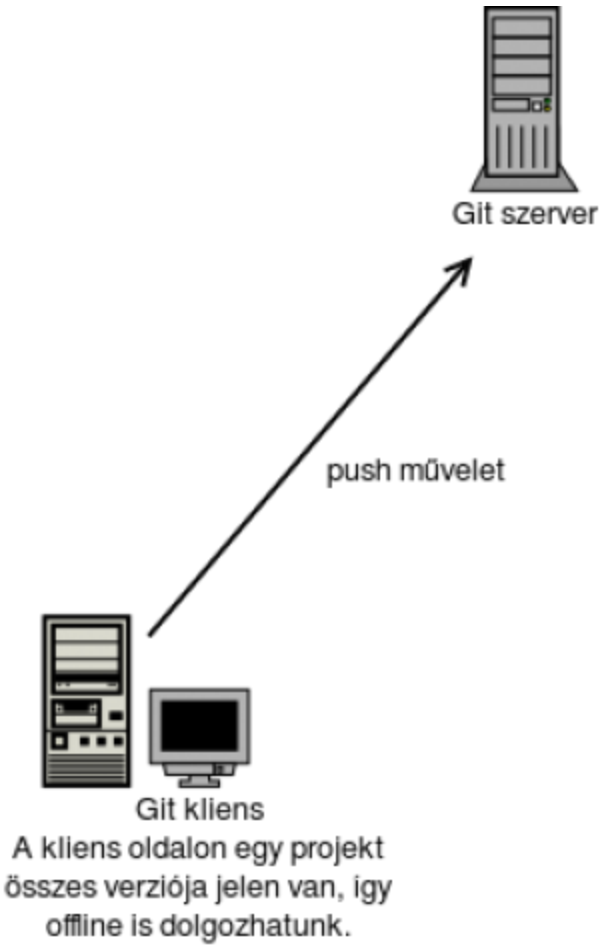
A távoli tároló, ahova mindenki feltölt.

downstream

A helye tároló ahol dolgozol.

Kliens-szerver felépítés

A git szerver-kliens alapú, de használható szerver nélkül is, mivel kliens oldalon is tárolódik az összes állomány. Alapveten egy helyi tárolóban dolgozunk, a szerverre egy push mvelettel töltjük fel az adatainkat.



A letöltés, clone, pull és fecth mveletekkel lehetséges.

Fájlok állapota

A fájlok a munkakönyvtárunkban alapveten két állapota lehet:

- untracked - nem követett
- tracked - követett
    - modified - módosult a commit óta
    - unmodified - a commit óta nem módosult
    - staged - a legközelebbi commit esetén tárolásra kerül

Kezd lépések

Bemutatkozás a Git számára

A git számára meg kell adni egy teljes nevet, és egy e-mail címet. Ez a név és cím fog megjelenni, minden kommitban.

```
git config --global user.name "Keresztnév Vezetéknév"
git config --global user.email "joska@zold.and"
```

Ezek az adatok a ~/.gitconfig nev fájlban kerülnek tárolásra, vagyis az adott felhasználónak glibális lesz, vagyis minden projekthez ezt fogja használni.

Állítsuk be a kedvenc szövegszerkesztnket.

```
git config --global core.editor mcedit
```

Állítsuk be színes kiíratás is:

```
git config --global color.ui true
```

Lekérdezés:

```
git config user.name
```

Beállítások lekérdezése:

```
git config --list
```

A lehetséges kimenete:

```
user.email=joska@zold.and
user.name=Nagy Jozsef
giggle.main-window-maximized=false
giggle.main-window-geometry=700x550+1192+459
giggle.main-window-view=FileView
push.default=simple
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://github.com/joska/myprojekt
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

Helyi munka

Új projekt

```
mkdir TesztProjekt
cd TesztProjekt
git init
```

A git init parancs létrehoz a TesztProjekt könyvtárban egy .git könyvtárat. Ez tulajdonképpen egy helyi adatbázis, ami eltárolja a projektünk verzióit, és minden azzal kapcsolatos információt.

Azt is mondhatjuk, hogy a projektünk, most már a Git része.

Használat

Kérdezzük le a fájlok státuszát:

```
git status
```

Most hozzunk létre egy állományt:

```
touch readme.txt
```

Kérdezzük le újból a státuszt:

```
git status
```

Tegyük követett állapotba a readme.txt fájlt:

```
git add readme.txt
```

Innentl kezdve commit esetén ez az állomány is tárolásra kerül.

Most nézzük meg újra a státuszt:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   readme.txt
#
```

A Changes to bi commited után felsorolt állományok staged állapotban vannak.

Változtassuk meg a readme.txt állományt:

```
echo "egy" > readme.txt
```

Ellenrizzük a státuszt:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   readme.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   readme.txt
#
```

Adjuk hozzá újra readme.txt fájlt:

```
git add readme.txt
```

Nézzük meg a státuszt:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   readme.txt
#
```

Fzzünk megint valamit readme.txt fájlhoz:

```
echo "ketto" >> readme.txt
```

Nézzük meg a változásokat a diff paranccsal:

```
git diff
```

A diff parancs csak akkor fog látni egy állományt ha az nincs staged állapotba.

Ha a staged állapotú állományokat is szeretnéd látni:

```
git diff --cached
```

Commit

A staged állapotú állományok tárolásra kerülnek.

```
git commit
```

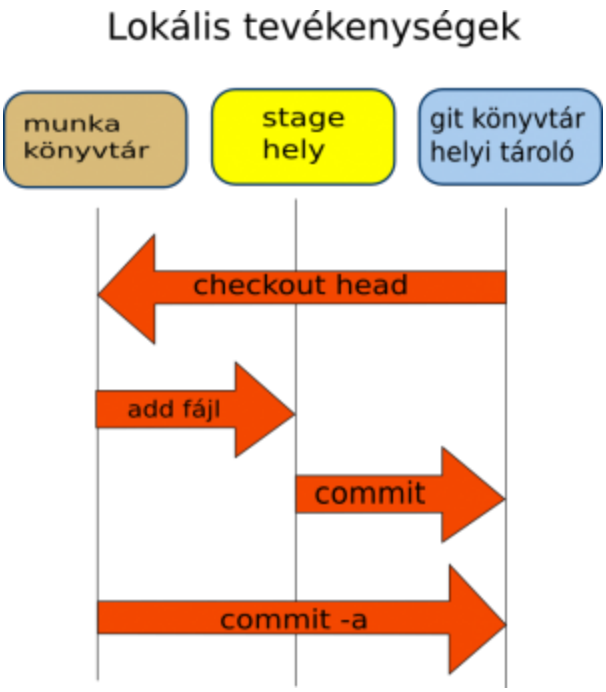
A git megnyitja egy szövegszerkesztt a megjegyzés leírásához.

A szövegszerkeszt beállítása:

```
git config --global core.editor mcedit
```

Persze mi magunk is megadhatjuk a parancssorban a megjegyzéseket:

```
get commit -m "Egy és kett hozzáadása"
```



A kommit módosítása:

```
git commit --amend
```

Ha szeretnénk a commit tulajdonosát és/vagy e-mail címét megváltoztatni:

```
git commit --amend --author="Vezetéknév Keresztnév <felhasznalonev@tartomanynev.hu>"
```

Ügyeljünk arra, hogy a beállított szövegszerkesztvel meg fog nyílni a commit szövege. Az alapértelmezett szövegszerkeszt pedig Windowson is a vim. Állítsuk be a jegyzettömböt, ha nem megy a vim használata. Ha megnyílt a commit és azt nem akarjuk szerkeszteni (csak a nevet és az e-mail címet akartuk változtatni), akkor csak simán lépjünk ki belle. Kilépés vimbl mentés nélkül:

```
<Esc>:q!<Enter>
```

A név és az e-mail cím megváltozik.

Tegyük fel, hogy korábbi commitot szeretnénk változtatni. A következ commitok vannak:

- egy
- ketto
- harom
- negy
- ot
- hat

A HEAD a „hat” commitra mutat, mi pedig szeretnénk megváltoztatni a „harom” és a „negy” commitot. A következket tegyük:

Adjuk meg, hogy a „ketto” után szeretnénk javítani:

```
git rebase -i ketto
```

A megadott szövegszerkesztben a commitokat látjuk felsorolva. Jelöljük meg az „harom” és a „negy” kommitokat szerkesztésre az „edit” szóval. Mentsünk, lépjünk ki a szerkesztbl. Végezzük el a változtatást:

```
git commit --amend --author="Vezetéknév Keresztnév <felhasznalonev@tartomanynev.hu>"
```

Lépjünk tovább:

```
git rebase --continue
```

Most a „negy” commitnál vagyunk. Végezzük el a változtatásokat:

```
git commit --amend --author="Vezetéknév Keresztnév <felhasznalonev@tartomanynev.hu>"
git rebase --continue
```

Elkészült.

A commit paranccsal a helyi tárolóba kerültek a fájlok. Nézzük meg mik vannak tárolva:

```
git ls-files
```

Esetleg:

```
git ls-tree --full-tree -r HEAD
```

Napló

```
git log
```

A commit-ok adatait tekinthetjük meg.

```
git log --abbrev-commit
```

```
git log --oneline
```

```
git log --no-notes
```

```
git log --pretty="format:%h %s %d"
```

A kimenet hasonló lehet:

```
5345711 Els sor beírva  (HEAD, master)
d78adf3 Kezdés
```

Fájlok törlése

```
git rm readme.txt
```

Ezzel a paranccsal töröljük fizikailag az állományt, töröljük a stage-rl és a tracedek közül.

Ha nem szeretnénk törölni a stage-rl és a tracedek közül, akkor használjuk a --cached kapcsolót.

```
git rm --cached readme.txt
```

Utolsó két commit:

```
git log -2
```

Commit-ok közötti különbségek:

```
git log -p
```

Fájlok átnevezése

Úgy szeretnénk átnevezni, hogy az verziókezelve legyen.

```
git mv eredetiNev ujNev
```

Fájlok kizárása a Gitbl

Néhány állományt szeretnénk majd kizárni a Git tárolónkból. Ez úgy tehetjük meg, ha felvesszük a következ állományba:

```
.gitignore
```

Ezek nem jelölhetk meg untraced állapotúként és nem lehetnek commitólni sem.

Példa:

.gitignore*.class # Csomagfájlok # *.jar *.war *.ear *.old

Linux alatt több információ:

```
man gitignore
```

A .gitignore állományt adjuk hozzá a tárolóhoz.

```
git add .gitignore
```

Módosítás különbségei

Nézzük meg az utolsó és az utolsó eltti kommit különbségét. Kommit után már csak a kommitok azonosítójának megadásával tudjuk megtenni.

```
git log -2
git diff 2343..3432
```

Ha még nem volt kommit, akkor elég a git diff:

```
git add fajlnev.txt
git diff
```

Ha csak egy behúzás volt a módosítása, de soron belül egyébként semmi nem változott, akkor is változásnak fogja fel:

```
-eredeti sor
+     eredeti sor
```

A -w kapcsolóval, kizárhatjuk a whitespace karakterek figyelését:

```
git diff -w 2343..3432
```

Két külön ág közötti különbség:

```
git diff master..devel-1
```

Ki változtatott

Lekérdezhetjük, az adott fájlban adott sort ki változtatta meg. Ezt a blame alparanccsal tehetjük meg.

```
git blame readme.txt
```

Munka a távoli szerverrel

Helyi változat létrehozása

A clone parancsot akkor használjuk, ha olyan tárolóval szeretnénk dolgozni, amelyik nem áll rendelkezésre a helyi gépünkön.

Ha szeretnénk dolgozni projekt1.git projekten, akkor a clone paranccsal töltsük le:

```
git clone 192.168.5.103:/home/git/projekt1.git
```

A javításokat visszatölthetjük, a push paranccsal.

A felhasználó a saját könyvtárára így is hivatkozhat:

```
git clone ssh://usernev@servernev/~/projekt1.git
```

```
git clone usernev@servernev:/~/projekt1.git
```

Név URL összerendelés

Ahhoz, hogy a távoli gépre fel tudjunk tölteni létre kell hoznunk egy nevet és egy hozzátartozó URL-t.

Az elsdleges név szokásosan az „origin” szokott lenni.

Hogy milyen összerendelések vannak, a következ képen kérdezhetjük le:

```
git remote
```

A következben egy ilyen hozzárendelést látunk:

```
git remote add origin ssh://usernev@servernev/~/repo/projekt1.git
```

Most megint nézzük meg milyen összerendelések vannak:

```
git remote
```

Ha használjuk a -v kapcsolót is, akkor azt is kiírja milyen URL tartozik az adott névhez.

Ha szeretnénk megváltoztatni egy nevet akkor a „rename” alparancs segít:

```
git remote rename origin masiknev
```

Törlés:

```
git remote rm masiknev
```

URL javítása:

```
git remote set-url origin ssh://usernev@servernev/~/repo/projekt1.git
```

Az URL változtatása másként:

```
git remote set-url origin git@192.168.5.103:/home/git/zold.git
```

Tartalom frissítése

Szeretnénk egy üres könyvtárunkban egy távoli gépen lév tárolóból dolgozni.

A git parancsnak útvonalban kell lennie.

Lépjünk a munkakönyvtárba:

```
cd c:\tervek\zold
```

Létrehozunk egy helyi tárolót:

```
git init
```

Beállítjuk a helyi tárolóban az elérési utat:

```
git remote add origin git@192.168.5.103:/home/git/zold.git
```

Leszedjük a tároló tartalmát:

```
git pull origin master
```

A pull parancsot, akkor szoktuk használni, ha rendelkezésünkre áll egy .git könyvtára. Ha nincs .git könyvtárunk, vagyis nincs munkaanyagunk, akkor git clone parancsot használjuk. A git pull parancs felülírja a helyi állományokat.

Tartalom letöltése megtekintésre

A git fetch paranccsal úgy tölthetjük le az állományokat, hogy nem írja felül a helyi állományokat. Egy külön branchként töltdik le.

```
git fetch
```

A git diff paranccsal megnézhetjük a különbséget:

```
git diff master..origin/master
```

Ha szerverrl leszedett állományok megfelelnek, akkor összeolvaszthatjuk a helyi állományokkal:

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

Feltöltés

Ha git clone paranccsal szedtük le a munkaanyagot, akkor elég a git push parancs:

```
git push
```

```
git push origin master
```

A --tags kapcsolóval, a címkék is feltöltésre kerülnek, ha vannak.

```
git push --tags origin master
```

Egyszerre két branch is feltölthet:

```
git push origin master devel
```

Mit tartalmaz a távoli tároló?

A show parancs közbeiktatásával azt is megtudhatjuk mit tud rólunk a távoli szerver.

```
git remote show origin
```

gitk

A gitk egy git tároló böngész.

Megmutatja a tárolóban történt változásokat, a kiválasztott kommitokat. A gitk ezeket vizuálisan is megjeleníti, miközben megmutatja a kommitok kapcsolatát.

Egyik gyakran használt kapcsolója a --all, amely az összes változat megmutatására utasít. Ha ezt nem használjuk, akkor csak az aktuális változat, például a master ágat mutatja.

Távoli elágazások

```
git branch -a
```

Kezdpont ez legyen:

```
git checkout -b fejlesztes remotes/origin/fejlesztes
```

Helyben is létrejön és aktív is lesz.

Elágazás

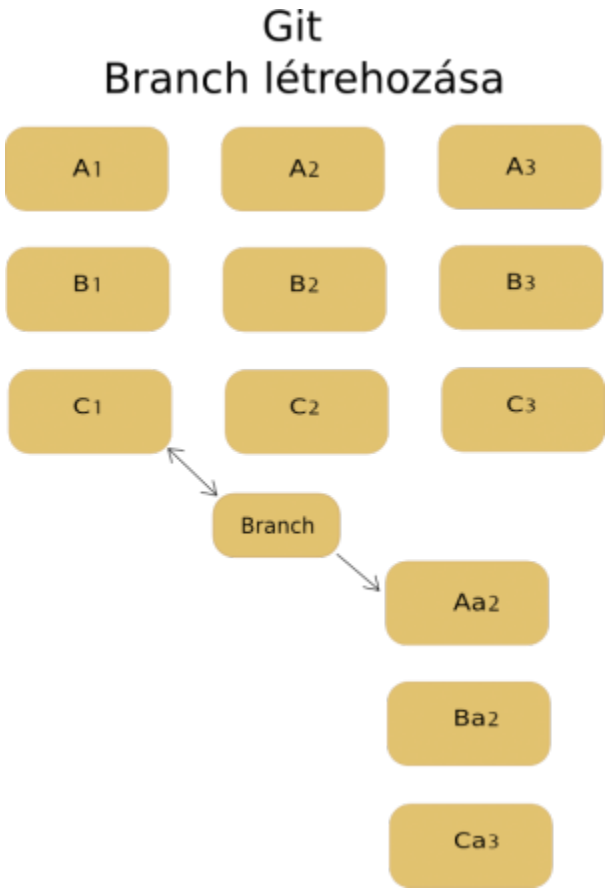
Elágazás létrehozása

Az elágazás a Git-ben a branch. A branch alparanccsal tudunk egy külön elágazás létrehozni.

```
git branch ujagneve
```

A parancs csak létrehozza a branch-et, de a Head nem áll át.

A példa kedvéért legyen egy projekt, amely három állományból áll, A1, B1 és C1. Lásd az ábrán:



A továbbfejlesztés eredményeként, létrejön a A2, B2 és C2, megint továbbfejlesztve A3, B3 és C3. Ha létrehozok egy branchet, akkor egy külön ágon másolat jön létre a három állományból. A képen az els kommit után egy branchet is készítettünk, amely egy külön ágat alkot.

A létez branchek **listázásához** írjuk a parancssorba:

```
git branch
```

Ha létrehoztunk egy devel-1 branchet, akkor ehhez hasonlót kell lássunk a kimenetben:

```
devel-1
* master
```

A * karakter mutatja az aktív branchet.

Átállás másik elágazásra

Másik branch-re – elágazásra – átállás a checkout alparanccsal lehetséges:

```
git checkout ujagneve
```

A kiadott parancsok ez után a ujagneve nev branch-on hajtódnak végre.

Helyi és távoli branch

A helyi és távoli branchet a -a vagy --all kapcsolókkal tekinthetjük meg:

```
git branch -a
```

Branch példa

Hozzunk létre egy devel-1 branchet:

```
git branch devel-1
```

Hozzunk létre egy ficsor-1 branchet:

```
git branch ficsor-1
```

Hozzunk létre egy kiserlet-1 branchet:

```
git branch kiserlet-1
```

Hozzunk létre egy fejlesztes-1 branchet:

```
git branch fejlesztes-1
```

Hozzunk létre egy sajatsag-1 branchet:

```
git branch sajatsag-1
```

Álljunk át a devel-1 elágazásra:

```
git checkout devel-1
```

Álljunk vissza a fágra:

```
git checkout master
```

A -b azonnal létre hozza a devel-2 ágat, ha az nem létezik:

```
git checkout -b devel-2
```

Branch törlése

Töröljük a branchet:

```
git branch -d devel-01
```

A nyomai megmaradnak.

Nyomtalanul törlés:

```
git checkout master
git branch -D devel-01
```

A -D esetén nem ellenrzi, hogy volt-e merge, ezért óvatosan.

Branch átnevezése

```
git branch -m devel-01 ficsor-01
```

Branchek összefésülése

Adott a master és a fejlesztes branch. Szeretnénk a masterbe fésülni a fejlesztes branchet. Átváltok a master ágra, majd összefésülök:

```
git checkout master
git merge fejlesztes
gitk
```

A fast forward az jelenti problémamentesen össze lett fésülve. Ilyenkor a gitk-ban látszik hogy a két branch azonos.

Konfliktus esetén szerkesszük a fájlt és töröljük a >>>>, =====, >>>>> sorokat. Majd kommit.

Segítség

A git help parancs önmagában is segít. Kiírja milyen alparancsai vannak a gitnek:

```
git help
```

Ezek után már könny megnézni egy alparancsról mit kell tudni:

```
git help <parancs>
```

Néhány alternatíva:

```
git <parancs> --help
man git-<parancs>
```

Például:

```
git help config
```

Az alparancsokról kézikönyvet a Debian GNU/Linux rendszereken így is kérhetünk:

```
man git <alparancs>
```

git gui

A git rendelkezik egy gui nev alparanccsal is. Ekkor beindul egy grafikus felület, ahol git mveleteket kattintgatva tudjuk végrehajtani.

Egyszeren írjuk be parancssorba:

```
git gui
```

Egyéb beállítások

Ékezetes fájlnevek:

```
git config --global core.quotepath false
```

Automatikus színezés:

```
git config --global color.ui auto
```

Álnevek létrehozása:

```
git config --global alias.co "checkout"
git config --global alias.gl "git log"
```

CR cseréje CRLF-re:

```
git config core.autocrlf true
```

Értéke true, false vagy input lehet.

```
core.autcrlf=true

       repo
      /    \
     /      \
crlf > lf   lf>crlf


core.autcrlf=input

       repo
      /    \
     /      \
crlf > lf    \


core.autcrlf=false

       repo
      /    \
     /      \
    /        \
```

Címkézés

Az egyes kommitok azonosítói elég barátságtalan egy átlagember számára. Lehetségünk van az egyes kommitokhoz egy címkét rendelni, így az azonosító szám helyett, használhatjuk ezt a címként. Címkét a már végrehajtott kommithoz rendelhetünk. A kommit után csak írjuk be:

```
git tag cimke01
```

Ellenrizzük a gitk paranccsal:

```
gitk
```

A használatban lév címkéket lekérdezhetjük ha a tag alparancsnak nem adok meg paramétert:

```
git tag
```

Újabb kommit után a címke az elz kommithoz fog tartozni. Például:

```
echo "új sor" >> vmi.txt
git add vmi.txt
git commit
gitk
```

A címkék lehetvé teszik a könnyebb állapotváltást. Például:

```
git checkout cimke01
gitk
```

Itt még visszaválthatok az elz kommitra:

```
git checkout master
```

Az újabb commithoz adjunk egy újabb címkét.

```
git tag cimke02
gitk
```

A diff parancs is könnyebben használható címkékkel:

```
git diff cimk01..cimke02
```

A címke a törölhet a következ módon:

```
git tag -d cimke02
```

Kommitok eldobása

Ha még nem volt commit, de a változásokat szeretném eldobni:

```
git reset --hard
```

Az utolsó kommit óta eltelt minden változat eldobva.

```
git status
```

A fájlból is eltünik.

A hashre azonosítóra hivatkozva, bármely kommit eldobható.

Megnézem a naplót:

```
git log
```

Kikeresem azt az utolsó kommitot amit még szeretnék megtartani.

```
git reset --hard 3483fg34
```

A HEAD ezentúl ide fog mutatni.

Átmeneti mentés

```
git status
touch ujfajl.txt
git add ujfal.txt
git commit
```

Módosítom a fájlt:

```
echo körte > ujfajl.txt
git status
```

Ezt a fájlt szeretnénk ideiglenesen eltenni:

```
git stash save "változat egy"
git status
```

Újra a régivel folytatom.

```
echo szilva > ujfajl.txt
git stash save "változat kett"
git status
```

Két fájl átmeneti tárolóba került. Listázzuk ezeket a fájlokat:

```
git stash list
```

A kimenet ehhez hasonló lesz:

```
stash@{0}: On master: változat egy
stash@{1}: On master: változat kett
```

Részletesebb információ:

```
git stash show
```

Esetleg a -p paraméterrel a különbségek is megjelennek.

```
git stash show -p stash@\{0}
```

A másik fájl:

```
git stash show -p stash@\{1}
```

Módosítás alkalmazása:

```
git stash apply
```

A legutoljára elmentett kommitot alkalmazza.

```
git stash list
```

Fájl tartalmának ürítése:

```
git reset --hard
git stash list
```

Utoljára elmentett kommitot törli és alkalmazza:

```
git stash pop
```

Megadhatjuk melyik kommit legyen:

```
git stash pop stash@\{0}
git stash list
git status
```

Nézzünk bele a fájlba. Ott van.

Átmeneti tárolóból törlés:

```
git stash drop
```

Így, paraméter nélkül, az utolsót törli.

Archívum készítés

A munkakönyvtárról készíthetünk archívumot, adott formátumban. A hozzáférhet formátumok a -l vagy --list kapcsolóval kérdezhetk le.

```
git archive -l
```

LinuxMint rendszeren nálam a következ lehetségeket kapom:

```
tar
tgz
tar.gz
zip
```

Tar csomag készítése

```
git archive HEAD > ../csomag.tar
```

Vagy tömörítve:

```
git archive HEAD | gzip > ../csomag.tar.gz
```

Esetleg adjuk meg a formátumot:

```
git archive HEAD --format=tar.gz > ../csomag.tar.gz
```

Takarítás

Ellenrizzük, hogy sérültek-e az adatok:

```
git fsck
Checking object directories: 100% (256/256), done.
```

Ellenrizzük le, mennyi adatot foglal egy .git könyvtárunk:

```
du -sb .git
```

Ekkor bájtban megadva láthatjuk az eredményt. Például:

```
81706 .git
```

Ha Kibibájtban szeretnénk:

```
du -sh .git
```

Az eredmény ehhez hasonló:

```
160K
```

Ha felesleges objektumok vannak, egy egyszeren tömöríteni kellene, akkor használjuk a git gc parancsot:

```
git gc
```

A futás után egy lehetséges kimenet:

```
git gc
Counting objects: 3, done.
Writing objects: 100% (3/3), done.
Total 3 (delta 0), reused 0 (delta 0)
```

Ezek után ellenrizzük újra a könyvtár méretét:

```
du -sb .git
70834 .git
```

```
du -sh .git
148K .git
```

Összefésülés

merge

Legyen egy háromszög kerület, területszámító program, amelyet eddig fejlesztünk:

Program01.javaclass Program01 { public static void szamitHaromszog() { int 3 + 4 + 5; } public static void main(String[] args) { System.out.println("Hi"); } }

Ekkor csinálunk rá egy elágazást. Legyen a neve: jokod.

```
git branch devel
```

Ezt az ágat azonban nem fejlesztjük tovább. A master ágat azonban tovább fejlesztjük. Lesz egy jegy() metódus, majd kommit. Lesz egy ertek() metódus, majd kommit.

Program01.javaclass Program01 { public static void szamitHaromszog() { int 3 + 4 + 5; } public static void jegy() { System.out.println("Nevem jön ide"); } public static void ertek() { System.out.println("Értéke ez."); } public static void main(String[] args) { System.out.println("Hi"); } }

Rájövünk, hogy nem is így kellett volna. Áttérünk a devel változatra:

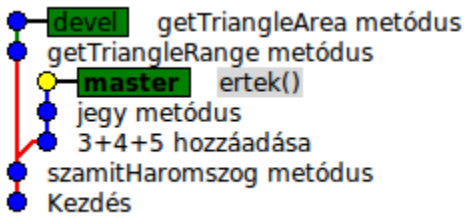
```
git checkout devel
```

Elször írunk egy getTriangleRange() metódust. Majd írunk egy getTriangleArea() metódust.

Program01.javaclass Program01 { public static double getTriangleRange(double a, double b, double c) { return a + b + c; } public static doubel getTriangleArea() { double s = (a + b + c) /2; return Math.sqrt(s*(s-a)*(s-b)*(s-c)); } public static void main(String[] args) { System.out.println("Hi"); } }

Ellenrizzük a gitk paranccsal hol tartunk:

```
gitk
```



A devel ág kódja teljesen jó. A master ág kódja viszont teljesen rossz amióta lett egy új branch. Átváltunk a master ágra:

```
git checkout master
```

Majd a merge paranccsal összefésülést kezdeményezünk:

```
git merge devel
```

A kimeneten kiíródik a konfliktus ténye:

```
Auto-merging Program01.java
CONFLICT (content): Merge conflict in Program01.java
Automatic merge failed; fix conflicts and then commit the result.
```

Az egyik megoldás lehet kézi javítás.

Ha most megnyitjuk a Program01.java állományt, akkor ezt látjuk:

Program01.javaclass Program01 { <<<<<<< HEAD public static void szamitHaromszog() { int 3 + 4 + 5; } public static void jegy() { System.out.println("Nevem jön ide"); } public static void ertek() { System.out.println("Értéke ez."); ======= public static double getTriangleRange(double a, double b, double c) { return a + b + c; } public static doubel getTriangleArea() { double s = (a + b + c) /2; return Math.sqrt(s*(s-a)*(s-b)*(s-c)); >>>>>>> devel } public static void main(String[] args) { System.out.println("Hi"); } }

Kitöröljük ettl: ««««< HEAD az egyenlség jelekig, majd a végérl töröljük a »»»»> devel sort. Elkészült.

A második, hogy visszavonjuk az összefésülést:

```
git merge --abort
```

A harmadik lehetség, hogy átvegyük a develágból az összes módosítást:

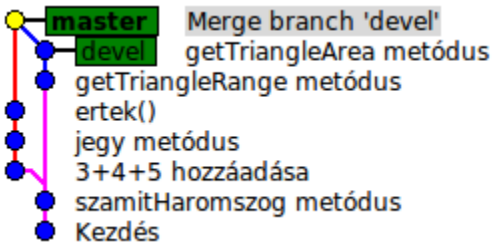
```
git checkout MERGE_HEAD -- .
```

Ez után jöhet a kommit:

```
git commit
```

Ellenrizzük a gitk paranccsal hol tartunk:

```
gitk
```

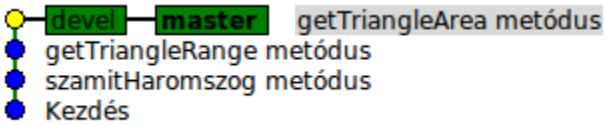


Majd törölhetjük a devel ágat:

```
git checkout master
git reset --hard devel
```

Ellenrizzük a gitk paranccsal hol tartunk:

```
gitk
```



Ezen a ponton a devel és a master ág megegyezik.

rebase

A példa kedvéért vegyünk egy egyszer példát: Feladatunk egy szamok.txt állomány fejlesztése, amely soronként tartalmaz a számokat. Az els sor egyeseket, a következ ketteseket, stb.

Indulás

```
echo 11111 > szamok.txt
git init
git add szamok.txt
git commit -m "Kezdés"
```

```
echo 222222 >> szamok.txt
git add szamok.txt
git commit -m "Kettesek"
```

Elágaztatunk, és a devel-1-et fejlesztjük tovább:

```
git branch devel-1
git checkout devel-1
echo "III III III III III" >> szamok.txt
git add szamok.txt
git commit -m "Hármasok"
```

```
echo "IV IV IV IV IV" >> szamok.txt
git add szamok.txt
git commit -m "Négyesek"
```

Egyesítjük a master és a devel-1 ágat:

```
git checkout master
git rebase devel-1
```