Git basics

Contents

Áttekintés

A git egyrészr?l szeretne teljesen elosztott lenne, de valójában ugyan úgy van egy "mester kópia" a szerveren ahogy az SVN-ben, amit mindenki lemásol magához és oda tölti vissza a változásokat.

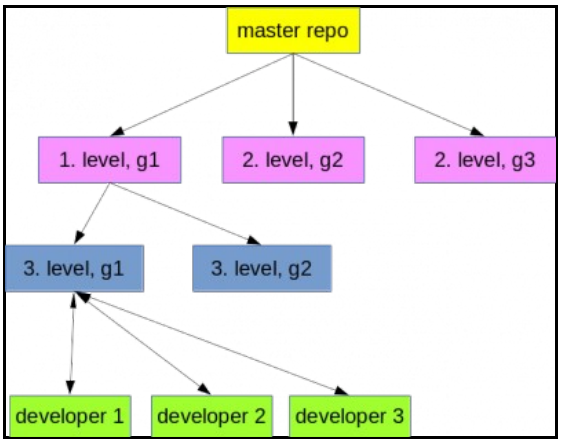
F?bb tulajdonságok

- Minden felhasználónál a teljes repository megtalálható. Ez azt jelenti, hogy mikor készítünk magunknál egy "másolatot" (szándékosan használtam itt a másolat szót), akkor valójában az egész repository-t lemásoljuk magunkhoz az összes branch-el, tag-el, változtatással együtt. Tehát anélkül hogy hozzá kéne férni a szerverhez, a lemásolás id?pontjáig az összes létez? lekérdezés lefuttatható mert minden adatunk megvan hozzá. Nyilván ha valaki már felvitte a változtatásit a szerverre, akkor el?bb nekünk frissíteni kell a helyi adatbázisunkat, tehát ez szintén csak egy látszat el?ny, egy átlagos méret? projektben ennek szerintem semmi haszna. Másrészr?l ha nagyon nagy a projekt, akkor a lokális tárhely szükséglet is gondokat okozhat. Azt mondják, hogy ez az (ál)decentralizáltság azért is jó, mert több helyen megvan pont ugyan az. Ez azért kicsit ferdítés, mert egy valódi szerveren az adatbiztonság mindig meg van oldva, az adatvesztés nem fordulhat el?. Nem is beszélve arról, hogy akárcsak az SVN-nél, a lokális kópiák általában nem a legfrissebbek, a teljes kép mindig csak a szerver kópián van meg.

- Az egyes felhasználóknál lév? lokális kópiák teljes érték? verzió követ? rendszerek. Ugyan úgy "commit"-álni kell a lokális kópiánkba, mint az SVN-nél a távoli szerverre. Ez pl nagyon hasznos lehet valakinek, aki csak egy lokális verzió követ?t akar felrakni magához gyakorlatilag egy parancs kiadásával. Ha akarjuk, innen lehet feltolni egy másik repozitory-ba, tipikusan a szerverre.

- A branchek közötti váltás csak egy mozdulat: Mivel nálunk a teljes kópia (legalább is annak valamilyen id?pillanatban létez? változata) ezért egy paranccsal át lehet állni egy adott ágra. Nagy különbség még, hogy egyszerre csak egy ág aktív, amin éppen dolgozunk. Az ágak közötti lokális merge nagyon egyszer?. Ide - oda "beleolvaszthatjuk" a forráskódunkat egyik ágból a másikba, majd bármelyik ágat feltolhatjuk a szerver repository-ba. Ez elég hasznos lehet.

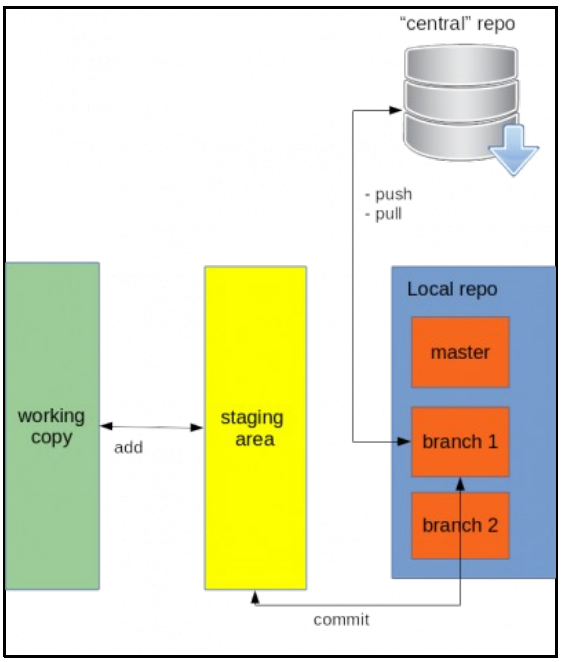
- A repository-k hierarchiába szervezhet?ek. Valójában erre lett kitalálva a git. Ez lehet?vé teszi akár több ezer együtt dolgozását ugyan azon a giga projekten. Szerintem itt jön el? a git igazi el?nye, több ezer ember nehezen használna egyszerre SVN alapú projekteket. Egy giga projektben több f? fejleszt?i csoportot kell kialakítani a f? feladatok mentén. Majd azokat további kisebb csoportokra lehet osztani. Git-ben csinálunk egy legfels? repot, ez lesz a legfels? szint? "mester" repo, ezt lemásolják második szint? "mester" repók. Ezt használják a f? fejleszt?i csoportok. Nevezzük ezt "második szint? másolatoknak." Ezen repoknak csak olvasási joga van a mester példányhoz. A második szint? másolatokat lemásolják a harmadik szinten lév? fejleszt?i csoportok. Ezen harmadik szint? csoportok is csak olvasni tudják a második szintet. A harmadik szint? másolatokhoz már közvetlen fejleszt?k kapcsolódnak, oda tolják fel a fejlesztésüket. Az els? körös tesztelést önállóan végzik a 3. szinten lév? csoportok. Mikor kész vannak elküldik email-ben a változtatást a 2. szinten lév? szül?jüknek. A második szinten lév? felel?sök összegy?jtik a 3. szintr?l érkezett fejlesztéseket, és belerakják helyben a 2. szint? másolatokba. Ezek után 2. szinten is tesztelésre kerül a fejlesztés. Majd a fejlesztések végül elküldésre kerülnek a mester repóba.



Felépítés

A lokális git másolatunk három részre osztható.

- Lokális, verzió kontrollált másolata a fájloknak. Itt az összes ág és verzió megtalálható a lokális másolat utolsó frissítése óta. (A centrális repó-ban közben már lehet hogy tök más van)
- staging are: ez egy átmeneti tároló. Itt kell összegy?jteni azokat a fájlokat, amiket commit-álni szeretnénk a lokális repónkban. (általában felesleges)
- working copy: ez a jelenlegi munkaterület. Itt a lokális másolat valamelyik ága van berakva (brach). + itt vannak a még nem verziózott fájlok. Ez az ami megjelenik a fájlrendszerben, ha megnyitom a git mappáját a projektnek. Azt hiszem a git szimbolikus linkekkel dolgozik. Ezért tudja olyan gyorsan kicserélni a munkaterületet egy másik branch-re. Ha átállunk egy másik ág-ra, akkor a fájlrendszerben már a másik ág fájljai fognak látszani. Így az svn-nél megszokott mappa / branch szemlélet itt nem célravezet?.



Ha van egy üres lokális git repository-nk, akkor az új fájlokat els?ként az **add** paranccsal hozzá kell adni a staging területhez. Innen a **commit** paranccsal adhatjuk hozzá az éppen használt ág-hoz. Alapesetben ez a master, vagyis a f? ág. (trunk az SVN-ben).

A megfelel? branch (ág) lokális változatát a **push** paranccsal tolhatjuk fel a "központi" repository-ba (ugyan abba az ágba). (Akár hogy is nézzük, igazából ez nem egy decentralizált verzió követ? rendszer). A központi másolatból a **pull** paranccsal másolhatjuk át a kiválasztott ágat a mi lokális másolatunk megfelel? ugyan azon ágába.

Note
Fontos különbség még az SVN-hez képest, hogy ha valaki kiad egy **push** parancsot, és feltolja a lokális változtatásait a lokális repóba, akkor a git úgy veszi hogy az egész branch módosult

. Ha valaki egy másik felhasználó **push** parancsa után egy tök másik fájlt akar feltolni ugyan abba a branch-be, akkor csak akkor fogja megtenni, ha az új változtatásokat els?ként leszedi a **pull** paranccsal, lokálisan egyesíti a fájlokat, majd utána az egészet feltolja.

Támogatott protokollok

A felhasználók (kliensek) a központi példányt **ssh**, **http** vagy **git** protokollon keresztül érhetik el. A változásokat a **push** paranccsal tolhatják fel a központi példányba, és a **pull** paranccsal hozhatják le.

Mi az ssh protokollt fogjuk használni, mivel ha van ssh hozzáférés egy szerverhez, akkor azonnal használható minden további komponens nélkül. Tehát ssh esetén nem kell külön egy git damon fusson a szerveren mint az SVN esetében. A centrális példány is csak fájlok összessége, amihez ssh-val hozzáfér a lokális git kliens.

Note
Azt kell látni, hogy az svn-el ellentétben az esetek többségében nem fut egy git daemon a szerver gépen

A lokális git programunk az adott (fájl megosztást is támogató) protokollon keresztül eléri a szerveren lév? fájlokat. Tehát nincs más dolgunk, mint a klienseknek egy olyan URL-t biztosítani, ami az adott központi git repository mappájára mutat. A többit elintézi a git.

- HTTP: az apache-ban készíteni kell hozzá egy virtuális hosztot, kb negyed óra m?ködésre bírni. Az autentikációt az apache végzi, el?nye, hogy anonymous hozzáférést nagyon egyszer? vele biztosítani.
- SSH: ehhez semmit nem kell konfigurálni. Ha egy felhasználó ssh-n keresztül képes belépni, és hozzáférni a git központi repository fájljaihoz, akkor azonnal használni tudja azt, nincs szükség egyéb konfigurációra. Pont ezért ez a legelterjedtebb protokoll.
- GIT: ez a git saját protokollja, kifejezetten anonymous hozzáférésre készült az open source világ számára.

Repository létrehozás

Nem feltétlen kell központi git repository, azt is megtehetjük, hogy csak egy lokális repository-t inicializálni, de akkor nem tudunk együtt dolgozni másokkal. Ezért mi els?ként egy központi repository-t fogunk készíteni a szerveren, majd a **clone** paranccsal ezt fogjuk lemásolni a munkaállomásokra mint lokális másolatok.

Központi másolat

Álljunk bele abba a mappába ahol a központi másolatot ki szeretnénk alakítani a szerveren, majd a **git init** paranccsal inicializálhatjuk a centrális repository-t. Itt megint megemlíteném, hogy két speciális kapcsolót kell használni az **init** parancs mögött, amivel jelezzük, hogy ez lesz a **központi** (mester) példány.

- **--bare**: Ennek a hatására a szerveren egy speciális repository fog létrejönni, ahol nem lesz staging vagy working area, a fájlok binárisan lesznek tárolva, ember számára nem lesz olvasható a tartalom. Ez kifejezetetten szerver üzemmódra van, itt nem lehet használni a **commit** vag az **add** parancsokat.
- **--share**: Szintén az jelzi, hogy ez a szerver példány, amihez többen fognak hozzáférni, így a git a push paranccsal megkapott fájlok tulajdonosát speciálisan fogja beállítani, hogy a csoportból mindenki hozzáférjen.

Tehát, inicializáljuk

```
$ cd /home/adam/git/testRepo
$ git init --bare --shared
```

Ekkor létrejött a szerver git struktúra:

```
$ ll /home/adam/git/testRepo
total 32
drwxrwsr-x.  2 adam adam 4096 Sep 29 16:57 branches
-rw-rw-r--.  1 adam adam  126 Sep 29 16:57 config
-rw-rw-r--.  1 adam adam   73 Sep 29 16:57 description
-rw-rw-r--.  1 adam adam   23 Sep 29 16:57 HEAD
drwxrwsr-x.  2 adam adam 4096 Sep 29 16:57 hooks
drwxrwsr-x.  2 adam adam 4096 Sep 29 16:57 info
drwxrwsr-x. 13 adam adam 4096 Sep 29 17:59 objects
drwxrwsr-x.  4 adam adam 4096 Sep 29 16:57 refs
```

SSH-val az alábbi URL-n tudják a kliensek lemásolni a központi másolatot:

git clone ssh://<user>@<host>/<full path to git repository directory>

A mi esetünkben:

```
git clone ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/
```

Lokális másolatok

SSH kulcsok beállítása

Ahhoz hogy ne kérjen jelszavat minden egyes git m?velet, el kell helyezzük a titkos kulcsunkat a saját számítógépünkön, aminek a publikus párja a git-et futtató ssh szerveren van. Azonban a titkos kulcsunkat az ssh kliens a ~./ssh/id_rsa néven fogja keresni. Ha a git -hez való titkos kulcsot rakjuk ide, akkor nem fogunk tudni más szerverhez kulccsal kapcsolódni. Erre találták ki a ~./ssh/config fájlt, ahol fel lehet sorolni, hogy milyen hosztokhoz milyen user névvel és titkos kulccsal csatlakozzon az ssh kliens.

```
# touch ~./ssh/config
# chmod 600 ~./ssh/config
```

Majd az alábbi tartalmat helyezzük el benne minden egyes hoszt-hoz:

```
Host 192.168.10.121
IdentityFile /home/adam/auth/ssh/titkos_kulcs
User adam

Host hostname2.com
IdentityFile /home/adam/auth/ssh/titkos_kulcs2
User adam2
```

Warning
Fontos hogy a config fájlon és a kulcsokon is 600 jog legyen

Központi példány másolása

Álljunk bele a lokális gépen abba a mappába, ahol a git repository-k vannak, de ne hozzunk létre a most másolandó repónak egy saját mappát, a copy parancs ezt majd megteszi.

Adjuk ki a git clone parancsot. Ha nem kulcsos az azonosítás az ssh szerveren, akkor be kell írjuk a jelszavunkat minden egyes hozzáféréskor.

```
$ cd /home/adam/git
$ git clone ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/
Cloning into 'testRepo'...
adam@jenkins.mycompany.hu's password:
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 9 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

Ezzel a **/home/adam/git** mappában létrejött a repository nevével megegyez? **testRepo** mappa. Láthatjuk, hogy a szerver struktúrával ellentétben, most létrejött a .git mappa. Ha nem szerver módban hozzuk létre a repository-t, akkor létrejön a working area is, ami egy fajta virtuális fájlrendszer.

Git beállítások

Végezzük el az alap git beállításokat. A git a felhasználó home mappájában a **.gitconfig** fájlban tárolja a beállításokat. Ezt vagy kézzel írjuk közvetlenül, vagy a **git config --global** paranccsal.

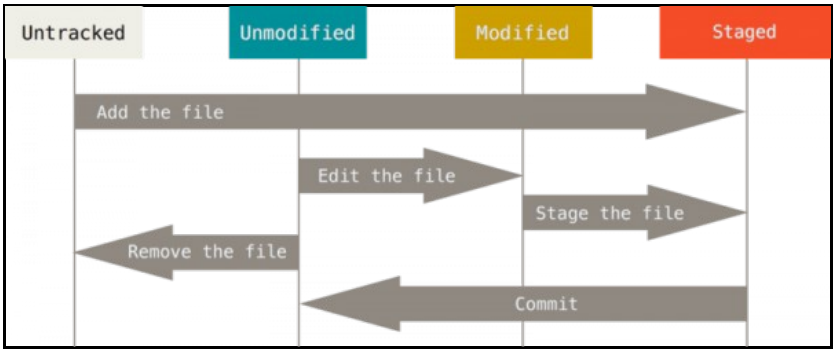
```
$ git config --global user.name "Berki Adam"
$ git config --global user.email berki.adam@mycompany.hu
$ git config --global core.editor mcedit
$ git config --global diff.tool meld
```

Ekkor a ~/**.gitconfig** fájlban az alábbi sorok jöttek létre:

```
[user]
        name = "Berki Ádám"
        email = berki.adam@mycompany.hu
[core]
        editor = mcedit
[diff]
        tool = meld
```

Git alap parancsok

Az alábbi képen láthatjuk, hogy mi történhet egy fájllal a lokális másolatunkban:



A munkaterületen egy fájl értelem szer?en, vagy még nincs hozzáadva a repository-hoz, vagy hozzá van már adva és módosítva van, vagy hozzá van már adva és nincs is módosítva.

STAGE

Fájl hozzáadása a stage területhez: **git add <filename>**

```
$ git add file1.txt file2.txt file3.txt
```

Ha vissza akarunk vonni egy fájlt a stage területr?l, akkor ezt a **git reset HEAD <file name>** paranccsal tehetjük meg.

COMMIT ügyek

A **commit** parancs a lokális repository-ba a stage területr?l rakja be a fájlokat (tehát csak azokat amiket már a stage-hez hozzáadtunk): **git commit -m 'commit üzenet'**

```
$ git commit -m 'Els? commit'
[master 8dd4892] Els? commit
 4 files changed, 3 deletions(-)
 delete mode 100644 elsofile.txt
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

```
 create mode 100644 file3.txt
```

Ha nem akarjuk külön hozzáadni a stage területhez a fájlokat, akkor a **-a** kapcsolóval mindent be lehet commit-álni, ami változott:

```
$ git commit -a -m 'comment'
[master 80e912f] comment
 4 files changed, 3 insertions(+)
 create mode 100644 file4.txt
```

Miel?tt a szerverre feltolnánk a repository jelenlegi tartalmát (push) vissza tudjuk vonni a legutolsó commitot: **git commit --amend**

Ha egy fájlt vissza akarunk állítani az eredeti állapotába: **git checkout -- <filename>**

status

A munkaterületen a fájlok állapotának kiírása: **git status -s**

```
$ git status -s
M  file1.txt
MM file2.txt
 M file3.txt
?? file4.txt
A  file5.txt
```

A git status-nak a kimenete a következ?t jelenti. <stage terület státusz><munkaterület státusz> <file neve>

- Tehát az els? oszlopban ha 'M' bet? van, azt jelenti, hogy módosult, de hozzá van adva a stage-hez.
- Ha a második oszlopban van 'M' bet?, akkor azt jelenit, hogy olyan módosítást tartalmaz, ami még a stage-hez nincs hozzáadva.
- Ha mind két oszlopban 'M' bet? van, azt jelenit, hogy már hozzá lett adva a stage-hez, de azóta módosult, tehát ha most nyomnánk egy commit-et, akkor csak az kerülne bele, amit a stage-hez adás el?tt beleírtunk.
- A '??' (tehát mind két oszlopban) jel azt jelenti, hogy új fájl, ami még nem volt hozzáadva a stage-hez sem.
- Az 'A' az els? oszlopban azt jelenit, hogy új fájl, de már hozzá van adva a stage-hez, de még sosem volt commitálva.

Ha a **-b** kapcsolót is belerakjuk a lekérdezésbe, akkor a brach infót is megmutatja:

```
$ git status -bs
## master...origin/master [ahead 1]
...
```

DIFF

Azt hogy a munkaterületen mi változott a **diff** paranccsal tudjuk megnézni. Ez a stage területen lév? fájlokat nem mutatja. Nem igazán ember barát a kimenet:

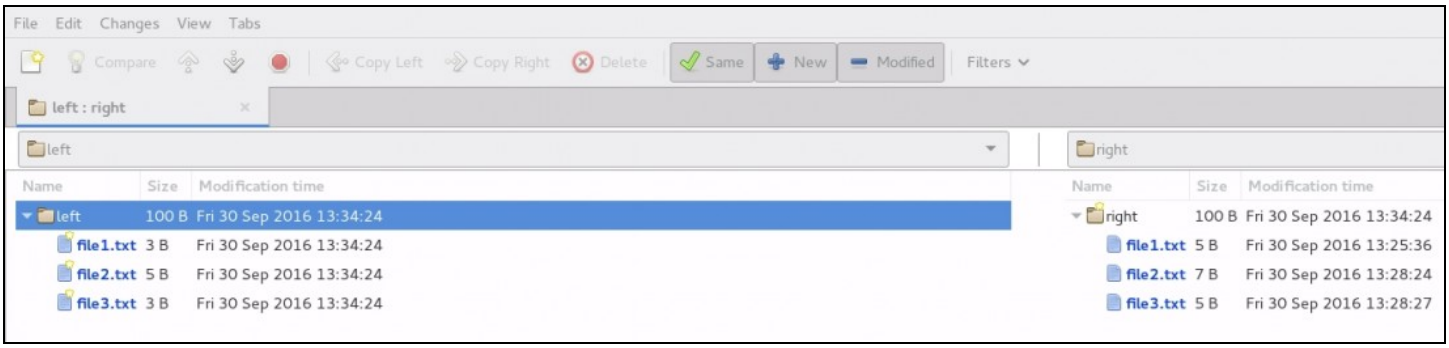
```
$ git diff
diff --git a/file1.txt b/file1.txt
index e7cb5c3..020c78b 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-mod
\ No newline at end of file
+moddd
\ No newline at end of file
```

Ha a **--staged** kapcsolót is mögé írjuk, akkor azt mutatja meg hogy milyen változás van a stage-en. (Gyakorlatilag ez a state tartalma)

Ha könnyen értelmezhet? kimenetet szeretnénk látni, akkor használjuk a meld-et. Ezt már korábban beállítottuk a ~/.gitconfig-ban mint alapértelmezett diff eszköz.

```
$ git difftool -y -d
```

- -y: ne kérdezzen rá
- -d: mappa szinten hasonlítson össze. Enélkül egyenként feldobálja az összes fájlt külön meld ablakban.



REMOVE

Az **rm** paranccsal a fájlt a teljes virtuális fájlrendszerb?l töröljük, tehát nem csak a lokális repository-ból, hanem a stage és a munkaterületr?l is el fog t?nni:

```
$ git rm file1.txt
error: the following file has local modifications:
    file1.txt
(use --cached to keep the file, or -f to force removal)
```

Ha csak verziózatlan fájlt akarunk bel?le csinálni a munkaterületen, akkor ezt az **rm --cached** kapcsolóval tehetjük meg. Ett?l még a munkaterületen meg fog maradni:

```
$ git rm --cached  file2.txt
rm 'file2.txt'
```

Commit history

A git log parancs megmutatja az eddigi teljes commit listát:

```
$ git log
commit 80e912f1c37ca3237b8248ad7ebb106aca17da59
Author: Berki Ádám <berki.adam@mycompany.hu>
Date:   Fri Sep 30 13:21:21 2016 +0200

    Második commit üzenet

commit 8dd4892161ced552752512c0a9b082c5c1dec6f6
Author: Berki Ádám <berki.adam@mycompany.hu>
Date:   Fri Sep 30 12:14:48 2016 +0200

    Els? commit üzenet
...
....
```

Rengetegféle képen lehet sz?kíteni a megjelenített listát.

Szerver ügyek

Távoli repók listája

Távoli elérések listázása (központi repository-k):

```
$ git remote -v
origin  ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/ (fetch)
origin  ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/ (push)
```

Külön kiírja, hogy melyik repóra milyen jogunk van (olvasás - fetch, írás - push)

Az els? oszlopban a kapcsolat neve látható. Ezzel kell rá hivatkozni, ha fel ill. le akarunk tölteni a szerverr?l. Jelen esetben csak egy távoli szerver van beállítva, aminek a neve **origin**, ami azt jelöli, hogy ez az a repó, amib?l klónoztuk a mi lokális példányunkat.

Egy adott kapcsolatról **remote show <név>** paranccsal kaphatunk részletesebb infót:

```
$ git remote show origin
adam@jenkins.mycompany.hu's password:
* remote origin
  Fetch URL: ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/
  Push  URL: ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/
  HEAD branch (remote HEAD is ambiguous, may be one of the following):
    adam2
    master
  Remote branches:
    adam1  tracked
    adam2  tracked
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (fast-forwardable)
```

Frissítés a szerveröl

Két lehet?ségünk van. A **pull** parancs minden változást letölt, és merge-l is a lokális repóba. A **fetch** parancs letölti a változásokat, de nem merege-li automatikusan, ezt majd nekünk kell megtenni.

```
$ git fetch
adam@jenkins.mycompany.hu's password:
```

ill.

```
$ git pull
adam@jenkins.mycompany.hu's password:
```

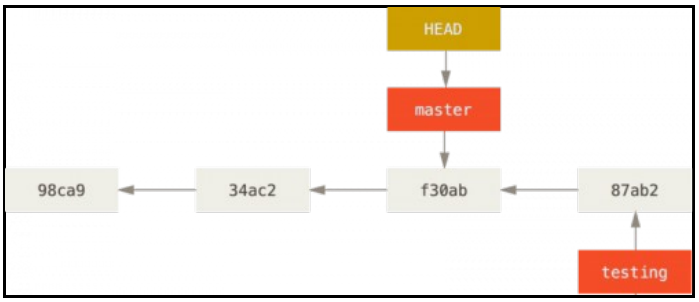
Mind két parancs mögött meg lehet adni a kapcsolat nevét (ha több távoli szerver is be van állítva, nálunk csak egy origin nev? van) és a branch nevét, amit fel akarunk tölteni. Ha nem adjuk meg egyiket sem, akkor nálunk az origin kapcsolatot fogja használni, valamit azt a branch-et, ami éppen a munkaterületre be van töltve. (jelenleg ez a master, ami SVN terminológiában a trunk)

git pull origin <branch neve>

Frissítés a szervere

Branch

Git-ben minden egyes commit egy pillanat felvétele a világnak. Ezen pillanat felvételek egymásra mutatnak. (Baloldalon van a legrégebbi, és ahogy haladunk el?re az id?ben mindig egy újabb commit lesz leglel?l, ami rá fog mutatni az el?z?re. Egy branch nem más mint egy kitüntetett mutató egy adott pillanat felvételre (narancssáraga). Ha egy branch-be kommitálunk, akkor egy saját commit láncot indítunk el arról pontról. A git úgy tartja nyilván az aktuálisan kiválasztott branch-et (ami be vat töltve a munkaterületre) hogy ráállítja a speciális HEAD mutatót. Igazából branch váltás közben nem csinál mást, mint hogy a HEAD mutatót átállítja egy másik branch-re, és az abban a snapshot-ban lév? fájlokat tölti be a munkaterületre.



A fenti képen az látható, hogy két branch-ünk van. A **master** branch (amit a git init magától létrehoz, hogy legalább egy branch legyen a repóban) és a **test** nev? branch. Jelenleg a munkaterületünkre a **master** van betöltve, mivel arra mutat a **HEAD** mutató. A test branch-be már volt egy commit a master hez képest.

Tip
A master is közönséges branch amit a git init hoz létre, és ráállítja a HEAD mutatót kapásból

Tip
A tag is egy speciális mutató, ami rámutat egy snaphost-re, de nem lehet mozgatni, vagy betölteni a munkaterületre (lásd lentebb).

Branch kezelése

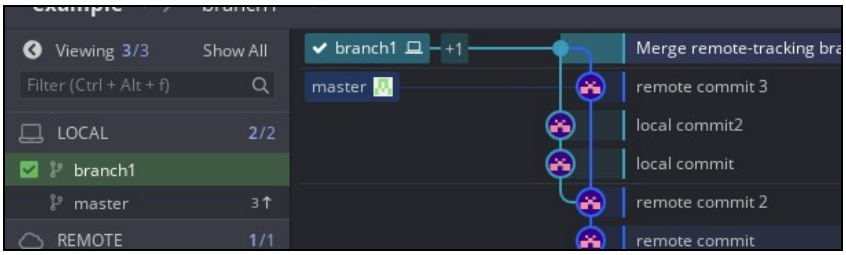
Létrehozás

Branch-et a **branch <ág neve>** paranccsal hozhatunk létre:

```
$ git branch adam3
```

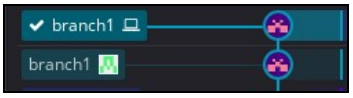
GitKraken-ben kétféle képen hozhatunk létre branch-et. A legegyszer?bb ha megnyomjuk a Branch gombot a fels? sávban:

Ennek a hatására létre fog hozni egy lokális branch-et azon a ponton, ahova a lokális head mutat, vagyis az aktuális lokális branch-en az utolsó commit-ra. A gomb megnyomására megjelenik egy input box amellett a commit mellett ahonnan a branch-et leágaztatjuk.

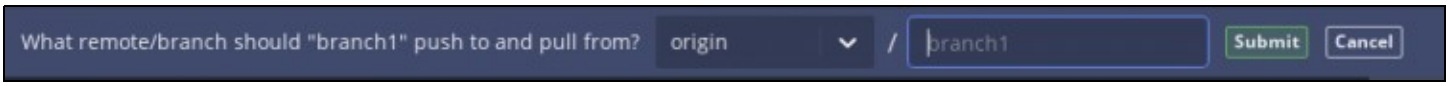


A fenti képen látható, hogy létrehoztam a **branch1** nev? branch-et a 'Merge remote tracking branch..' nev? commit-ból. Az új branch-et automatikusan checkout-olta a GitKraken. Ezt a baloldalon láthatjuk a 'LOCAL' szekcióban. A master fölött (a korábbi branch) zöld pipával szerepel a **branch1**. A commit tree-ben a branch1 név mellett egy +1 szerepel, ami azt jelenti, hogy egy másik lokális branch-nek is ide mutat a head-je. Ha fölé visszük az egeret láthatjuk, hogy a másik lokális branch a **master**.

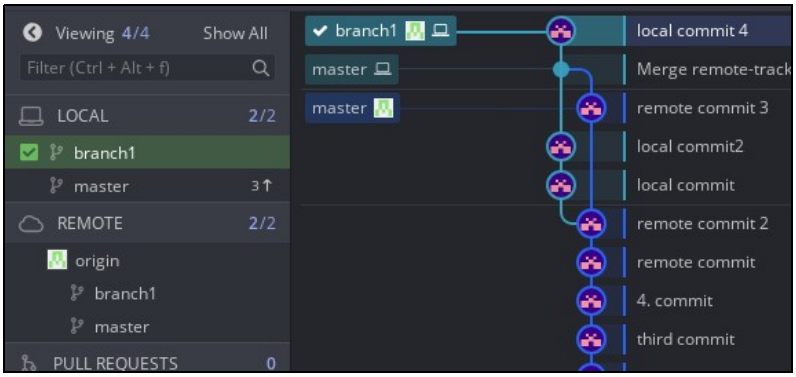
A branch neve mellette kis számítógép jelzi, hogy ezek a lokális branch head-ek.
Ha commit-álunk egyet az új lokális branch-en, akkor láthatjuk, hogy a korábbi master (amib?l kiindultunk) és az új branch1 head mutatója már egyel eltérnek egymástól:



Az is látszik a képen, hogy a **branch1** még a távoli repóban nem létezik, mert csak zöld kis számítógépes ikonja van. A master branch is egy commit-al el?rébb jár mint a távoli master.
A push megnyomásával juttathatjuk fel a lokális **branch1**-et a távoli repo-ba. Ekkor rá fog kérdezni, hogy mi legyen a távoli repo-ban a neve:



A push után láthatjuk, hogy a 'REMOTE' szekcióban is megjelent a 'branch1' és hogy a commit tree-ben a kis compute mellett megjelent a remote repo-t jelz? kis zöld ikon is:



A branch1 mellette kis pipa jelzi, hogy jelenleg a 'branch1' van check-out-olva.

Váltás

A **checkout** paranccsal válthatunk branch-et.

Warning
Fontos, hogy a munkaterületen nem lehetnek verziózatlan vagy módosított fájlok, mert a teljes munkaterület ki fog cserél?dni. Ezért amíg nincs mind a repóban, addig nem fogja hagyni a branch váltást

```
$ git checkout adam3
Switched to branch 'adam3'
```

Tip
A branch váltást tartják a git leger?sebb tulajdonságának, tényleg egy pillanat alatt lehet két branch között váltani, mivel a teljes másolattal rendelkezik minden kliens

Listázás

Jelenlegi branch listázásához a **git status** parancsot használhatjuk:

```
$ git status
On branch adam3
...
```

A branch-eket a git branch paranccsal listázhatjuk (paraméter nélkül)

```
$ git branch
  adam3
  testBranch
* master
```

A csillag azt a bracnh-et jelöli, ahol éppen állunk.

Listázzuk ebb?l csak azokat, akiket még nem mergel-tünk bele a jelenlegi branch-be (ahol most állunk)

```
$ git branch --no-merged
  testBranch
```

Note
A --no-merged paraméter csak azokat a bracnh-eket fogja listázni, amik nem ugyan arra a commit-ra mutatnak mint a jelenlegi branch

És listázzuk azokat akiket már mergel-tünk a jelenlegibe:

```
$ git branch --merged
  adam3
* maste
```

Note
A --merged azokat is mutatja, akiknek a mutatója pont arra a commit-ra mutat, ahol most állunk

Merge

Mindig úgy kezdjük a merge-t hogy beleállunk abba a branch-be, AHOVA mergelni akarunk, majd kiadjuk a **merge <branch neve AHONNAN>** parancsot. Két esetet kell megkülönböztetni.

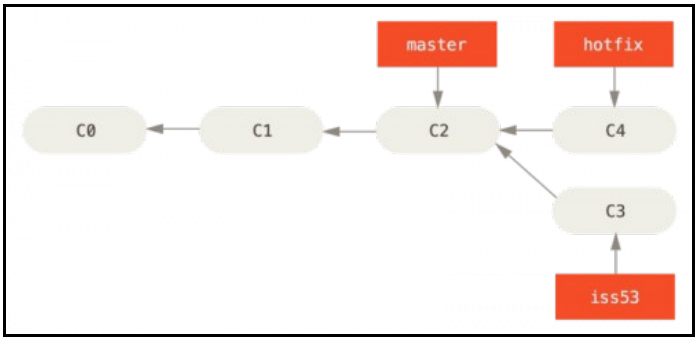
Az alább parancs a master-be merge-li az adam3 branch-et.

```
$ git checkout master
$ git merge adam3
```

Kétféle merge létezik:

Fast-forward merge

Ebben az esetben az a branch, ahonnan mergelni akarunk egyenes ági leszármazottja a jelenlegi branch-nek. Az alábbi ábrán pl a **hotfix**-et ha mergeljük a **master** -be.

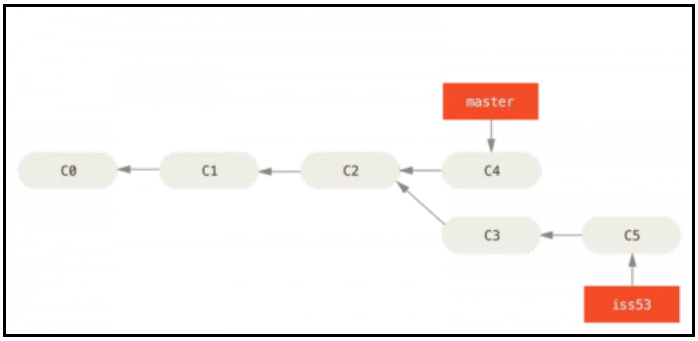


Ebben az esetben a git nem csinál mást, mint el?re mozgatja a master mutatóját (fast-forward). Persze ha volt változás, akkor a változásokat els?ként egyesíteni kell és az esetleges konfliktusokat feloldani. Ha fastForward merge történik, a git ezt jelzi:

```
$ git merge adam3
Updating f42c576..3a0874c
Fast-forward
...
```

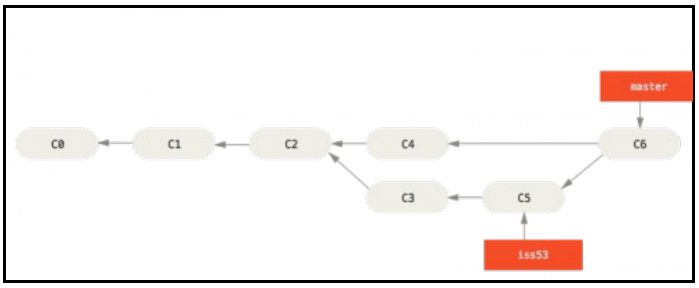
Két szül?s, 3 utas

Ebben az esetben az a branch ahonnan másolni akarunk (iss53), már nem közvetlen leszármazottja annak a branch-nek ahova merge-ni szeretnénk (master).



Ebben az esetben nem lehet egyszer?en el?re mozgatni a mutatót. A git meg fogja keresni a közös ?st, és egy három utas egyesítést fog csinálni a közös ?sb?l, a jelenlegi branch -b?l (a példában a master) és a bemásolandó branch-böl (a példában az iss53), és ebb?l az egészb?l létre fog hozni egy új commit-ot.

A mi példánkban a c6 lesz létrehozva a c5 c4 és c2 commit-okból.



Fontos, hogy az új merge commit elkészítéséhez a közös ?st is felhasználja a git. Ez azért kell, hogy lássa hogy azon fájlokban ami mind a két ágon módosult, mi volt a kiindulási alap. Ha pl. az egyik ágon az els? sort töröltem, a másik ágon az utolsót, akkor látni fogja az eredeti alapján, hogy nem az egész fájl változott csak az els? és az utolsó sor, ez alapján el tudj készíteni az egyesített verziót.

Merge GitKraken-ben

Rebase

.... TODO ....

Konfliktus feloldása

Meld mint merge tool

Els?ként a ~/**.gitconfig** fájlba be kell állítsuk, hogy a meld-et használja mint merge eszköz, ezen felül, hogy a meld három ablakát hogyan töltse fel:

```
...
[merge]
    tool = meld
[mergetool "meld"]
    cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

Konfliktus el?idézése

Ha egy adott fájl részletbe mind két branch-ben módosítottak, akkor a git nem fogja tudni magától feloldani a konfliktust. Ezek a fájlok nem kerülnek automation commit-álásra a jelenlegi branch-hez.

Próbáljuk ki. Álljunk bele az **adam3**-as brach-be. Majd itt módosítsuk a file1.txt-t, majd commit-áljuk be:

```
$ git checkout adam3
Switched to branch 'adam3'

$ echo "branch után írtam bele" >> file1.txt

$ git commit -a -m 'comment az adam3-ban'
[adam3 7305eb6] comment az adam3-ban
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Most lépjünk át a **master** branch-be, majd ott is írjunk bele a file1.txt-be:

```
[adam@adamdell testRepo]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ echo "ezt a tag készítés után írtam bele" >> file1.txt

$ git commit -a -m 'comment a master-ban'
...
```

Most adjuk ki a merge parancsot, vagyis hogy az adam3-as branch-et merge-jük a master-be (ahol most állunk)

```
$ git merge adam3
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Láthatjuk, hogy a konfliktus van. A **git status** parancs további infókkal szolgálhat:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   file1.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Ha belenézünk a munkaterület virtuális fájlszerkezetébe láthatjuk, hogy megjelentek a szabványos konfliktus fájlok:

```
ll
total 28
...
-rw-rw-r-- 1 adam adam 119 Sep 30 15:31 file1.txt.BACKUP.7629.txt
-rw------- 1 adam adam   3 Sep 30 15:31 file1.txt.BASE.7629.txt
-rw------- 1 adam adam  49 Sep 30 15:31 file1.txt.LOCAL.7629.txt
-rw------- 1 adam adam  33 Sep 30 15:31 file1.txt.REMOTE.7629.txt
```

Konfliktus feloldása

Ki kell adjuk a **git mergetool** parancsot, ami az összes konfliktusban lév? fájlra meg fogja nyitni a meld-et három ablakos nézetben. Középen lesz a végeredmény, bal oldalon a saját, még jobb oldalon a mergelni kívánt branch-ben lév? változat. (így állítottuk be a .gitconfig-ban.

```
$ git mergetool
Merging:
file1.txt

Normal merge conflict for 'file1.txt':
  {local}: modified file
  {remote}: modified file
```

Ha becsukjuk a meld-et, akkor rá fog kérdezni, hogy sikeres volt e a merege:

```
file1.txt seems unchanged.
Was the merge successful? [y/n] n
merge of file1.txt failed
```

Ha mentés nélkül csukjuk be a meld-et akkor rákérdez, hogy készen vagyunk e. Ha itt **n**-t választunk, akkor semmi sem változik. A **gt mergetool**-al újra meg tudjuk nyitni a meld-et. Ha mentjük a végeredményt (közép? csík), akkor kérdés nélkül felszámolja a segéd fájlokat és egy .orig végz?dés? fájlban elmenti az eredeti fájlt.

Nézzük meg, hogy most a file1.txt-ben a merge közben összeállított tartalom van. Nézzük meg a státuszt:

```
$ git status -s
M  file1.txt
?? file1.txt.orig
```

Nincs más dolgunk, mint a file1.txt-t hozzáadni a stage területhez, majd nyomni egy commit-ot:

```
$ git add file1.txt
$ git commit -m 'mege után'
[master bb194c8] mege után
```

Stash

... TODO ...

- transitive verb: to store in a usually secret place for future use ?often used with away
- noun: something stored or hidden away

Tag

Git-ben a tag-ek egy adott branch egy adott pontjának a pillanatfelvételei. (Valójában akár csak az SVN-ben, a branch és a tag egy pillanatfelvétele a világnak, de a tag nem változik, a tag mutatója mindig ugyan arra a commit-ra fog mutatni, ezt nem lehet megváltoztatni git-ben.

Tip
A tag is egy ponter, ami egy commit-re, pillanat felvételre mutat rá, csak nem lehet módosítani

Kétféle tag van:

- Lightweight: ez egy ugyan olyan mutató mint a branch, csak fixen egy helybe marad, míg a branch további életet él.
- Annotated: Az egész pillanatfelvételt lemásolja (nem csak mutató), ellen?rz? kódot, aláírást készíthetünk hozzá, és rengeteg plusz paramétert elmenthetünk hozzá.

Tag készítése

Pehelysúlyú tag-et a **git tag <tag név>** paranccsal készíthetünk:

```
$ git tag v1.1
```

Ez egy pillanat felvétel a jelenlegi branch legutolsó commit-járólé.

Ha a -a, -s, -m kapcsolók valamelyikével kiegészítjük a parancsot, akkor **Annotated** tag fog készülni:

```
$ git tag -a v1.4 -m "ez a 1.4-es verzió commentje"
```

Tag listázása

Listházáshoz használjuk a git tag parancsot: $ git tag v1.1

Ha már látjuk melyik tag érdekel, akkor a git show <tag név> paranccsal részletes infót kaphatunk. Láthatjuk a tag paramétereit (ha annotált volt) és hogy melyik commit-ból jött létre:

```
$ git show v1.4
tag v1.4
Tagger: Berki Ádám <berki.adam@mycompany.hu>
Date:   Fri Sep 30 14:22:21 2016 +0200

ez a 1.4-es verzió commentje

commit 80e912f1c37ca3237b8248ad7ebb106aca17da59
Author: Berki Ádám <berki.adam@mycompany.hu>
Date:   Fri Sep 30 13:21:21 2016 +0200

    comment
...
```

Tag betöltése

A tag-be úgy lehet belenézni, ha csinálunk bel?le egy új branch-et. Ehhez a checkout parancsot kell meghívni -b kapcsolóval: **checkout -b <új branch name> <stource tag név>**

```
$ git checkout -b branchFromV1.4 v1.4
Switched to a new branch 'branchFromV1.4'
```

Tag-ek feltöltése

A tag-et a git push parancs nem tölti f?l. Ezt implicit meg kell adni: **git push <kapcsolat név> <tag neve, amit fel akarunk tölteni>**

```
$ git checkout -b branchFromV1.4 v1.4
Switched to a new branch 'branchFromV1.4'
[adam@adamdell testRepo]$ git push origin v1.4
adam@jenkins.mycompany.hu's password:
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (9/9), 735 bytes | 0 bytes/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To ssh://adam@jenkins.mycompany.hu/home/adam/git/testRepo/
 * [new tag]         v1.4 -> v1.4
```