https://github.com/FAQGURU/FAQGURU

Mi a parancs egy git commit-ra?

```
git commit -a
```

-a a parancssorban utasítja a git-et az összes módosított nyomon követett fájl új tartalmának commit-olására. Te tudod használni

```
git add <file>
```

vagy

```
git add -A
```

git commit –a parancs előtt, ha új fájlokat commit-olunk először

Mi a különbség a Git és az SVN között?

A Git vs SVN vita fő pontja ebben rejlik: a Git egy elosztott verzióvezérlő rendszer (DVCS), míg az SVN egy központosított verzióellenőrző rendszer.

Mi az a Git?

A Git egy elosztott verziókezelő rendszer (DVCS). Nyomon követheti egy fájl változását, és lehetővé teszi, hogy visszatérjen az adott változáshoz.

Elosztott architektúrája számos előnyt nyújt más verziókezelő rendszerekhez (VCS), például az SVN-hez képest. Az egyik fő előnye, hogy nem egy központi szerverre támaszkodik a projekt fájljainak összes verziójának tárolására.

Hogyan lehet visszavonni a Git legutóbbi commit-ját?

```
$ git commit -m "Something terribly misguided"
$ git reset HEAD~                              # copied the old head to .git/ORIG_HEAD
<< edit files as necessary >>
$ git add ...
$ git commit -c ORIG_HEAD                      # will open an editor, which initially contains
the log message from the old commit and allows you to edit it
```

Mi az a Git fork? Mi a különbség a fork, az ág (branch) és a klón között?

A fork a repository távoli, szerveroldali másolata, amely eltér az eredetitől. A fork valójában nem Git-koncepció, inkább politikai / társadalmi ötlet.

A klón nem fork; a klón valamilyen távoli repository helyi másolata. Klónozáskor a teljes forrás repository-t másolja, beleértve az összes előzményt és ágat.

Az elágazás (branch) egy olyan mechanizmus, amely a változásokat egyetlen repository-ban kezeli annak érdekében, hogy végül egyesítsék azokat a többi kóddal. A branch olyan dolog, amely egy repository-ban található. Fogalmilag a fejlődés egy fonalát jelenti.

Mi a különbség a "git pull" és a "git fetch" között?

A legegyszerűbben kifejezve: a git pull egy git fetch követ, amelyet egy git merge követ.

A pull használatakor Git megpróbálja automatikusan elvégezni a munkáját helyetted. Környezetérzékeny, ezért a Git merg-eli az összes pull-olt commit-ot abba az ágba, amelyben éppen dolgozik. A pull automatikusan merge-eli a kommit-ot anélkül, hogy először átnézné őket. Ha nem kezeli szorosan branch-eket, gyakran konfliktusokba ütközhet.

Amikor fetch van, a Git összegyűjti azokat a commit-okat a célágból, amelyek nem léteznek az aktuális ágban, és tárolja azokat a helyi repository-ban. Ez azonban nem egyesíti őket a jelenlegi ággal. Ez különösen akkor hasznos, ha naprakészen kell tárolnia a repository-t, de azon dolgozunk, amely meghibásodhat, ha frissíti a fájlokat. Az commit-ok integrálásához a master ághoz használja a merge-t.

Mi a különbség a "pull request" és az "branch" között?

A branch csak a kód külön változata.

A pull request az, amikor valaki átveszi az repository-t, létrehozza saját repository-t, elvégez néhány változtatást, majd megpróbálja egyesíteni az ágat (a változásokat a másik személy kódtárába helyezi).

Hogyan működik a központosított munkafolyamat?

A központosított munkafolyamat központi adattárat (repository) használ a projekt minden változásának egyetlen pontjaként. Az alapértelmezett fejlesztési ágat masternek hívják, és minden változtatást végrehajtanak ebben az ágban.

A fejlesztők a központi adattár klónozásával kezdik. A projekt saját helyi példányaiban fájlokat szerkesztenek és változtatásokat hajtanak végre. Ezeket az új végrehajtásokat helyben tároljuk.

A hivatalos projekt módosításainak közzétételéhez a fejlesztők a központi törzságat a központi adattárba (repository) tolják. Mielőtt a fejlesztő közzétehetné funkciójukat, be kell szerezniük a frissített központi commit-okat, és újra kell alapozniuk a változtatásokat.

Más munkafolyamatokhoz képest a központosított munkafolyamatnak nincs meghatározott pull request vagy elágazási (forking) mintája.

Frissítenie kell a helyi repókat. Milyen git parancsokat fog használni?

Ez két lépésből áll. Először a remote-ról az origin-ba fetch-eljük:

```
git fetch origin
```

Ezután a local-t és a master-t merge-eljük:

```
git merge origin/master
```

Vagy egyszerűen:

```
git pull origin master
```

Ha az origin az alapértelmezett remote és master az alapértelmezett branch, akkor elég a

```
git pull .
```

Vissza kell térnie egy korábbi commit-ra, és nem érdekel a legújabb változások. Milyen parancsokat használjon?

```
git log // lists the commits made in that repository in reverse chronological order
git reset --hard <commit-sha1> // resets the index and working tree
```

Mi az a "git cherry-pick"?

A git cherry-pick parancsot általában arra használják, hogy a repository-n belüli egyik ágból származó különféle commit-okat más ágra vezesse be. Gyakori használat a karbantartási ágtól a fejlesztési ágig történő előre- vagy visszakötés.

Ez ellentétben áll más módszerekkel, például a merge-el és rebase-el, amely általában sok commit-ot alkalmaz egy másik ágra.

```
git cherry-pick <commit-hash>
```

Mondja meg, mi a különbség a HEAD, a working tree és az index között a Git-ben?

A working tree/working directory/workspace a látott és szerkesztett (forrás) fájlok könyvtárfája.

Az index/staging area egyetlen, nagy, bináris fájl a /.git/index fájlban, amely felsorolja az aktuális ág összes fájlját, azok sha1 ellenőrző összegeit, időbélyegzőit és a fájl nevét - ez nem egy másik könyvtár, amely másolja a fájlokat benne.

A HEAD hivatkozás a jelenleg kijelölt branch utolsó commit-jára.

Mikor használjam a "git stash" -t?

A git stash parancs elviszi az el nem commit-olt változtatásokat (mind staged and unstaged), elmenti őket későbbi felhasználás céljából, majd visszavonja őket a munkapéldányról.

Feltételezzük:

```
$ git status
On branch master
Changes to be committed:
new file: style.css
Changes not staged for commit:
modified: index.html
$ git stash
Saved working directory and index state WIP on master: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage
$ git status
On branch master
nothing to commit, working tree clean
```

Az az egyetlen hely, ahol használhatjuk a stashing-et, ha felfedezzük, hogy valamit elfelejtettünk az utolsó commit során, és már elkezdtük dolgozni a következő dolgon ugyanazon az ágon:

```
# Assume the latest commit was already done
# start working on the next patch, and discovered I was missing something

# stash away the current mess I made
$ git stash save

# some changes in the working dir

# and now add them to the last commit:
$ git add -u
$ git commit --ammend

# back to work!
$ git stash pop
```

Hogyan lehet visszaállítani az előző commit-ot gitben?

Tegyük fel, hogy ez megvan, ahol C a HEAD és (F) a fájlok állapota.

```
   (F)
A-B-C
    ↑
  master
```

Változtatások áthúzása a commit-ban:

```
git reset --hard HEAD~1
```

Most B a HEAD. Mivel a --hard használta, a fájlok visszaállnak a B commit állapotára.

```
git reset HEAD~1
```

Most azt mondjuk Gitnek, hogy mozgassa a HEAD mutatót egy lekötéssel (B) vissza, és hagyja a fájlokat a jelenlegi állapotukban, és a git status parancs megmutatja a C-be ellenőrzött változásokat. Visszavonhatja a commit-okat, de elhagyja a fájlokat és az indexet

```
git reset --soft HEAD~1
```

A git status parancs megadásakor látni fogja, hogy ugyanazok a fájlok szerepelnek az indexben, mint korábban.

Magyarázza el a Forking Workflow előnyeit

A Forking Workflow alapvetően különbözik a többi népszerű Git munkafolyamattól. Ahelyett, hogy egyetlen kiszolgálóoldali adattárat használna „központi" kódbázisként, minden fejlesztőnek megadja a saját szerveroldali adattárát. A Forking Workflow leggyakrabban nyilvános nyílt forráskódú projektekben látható.

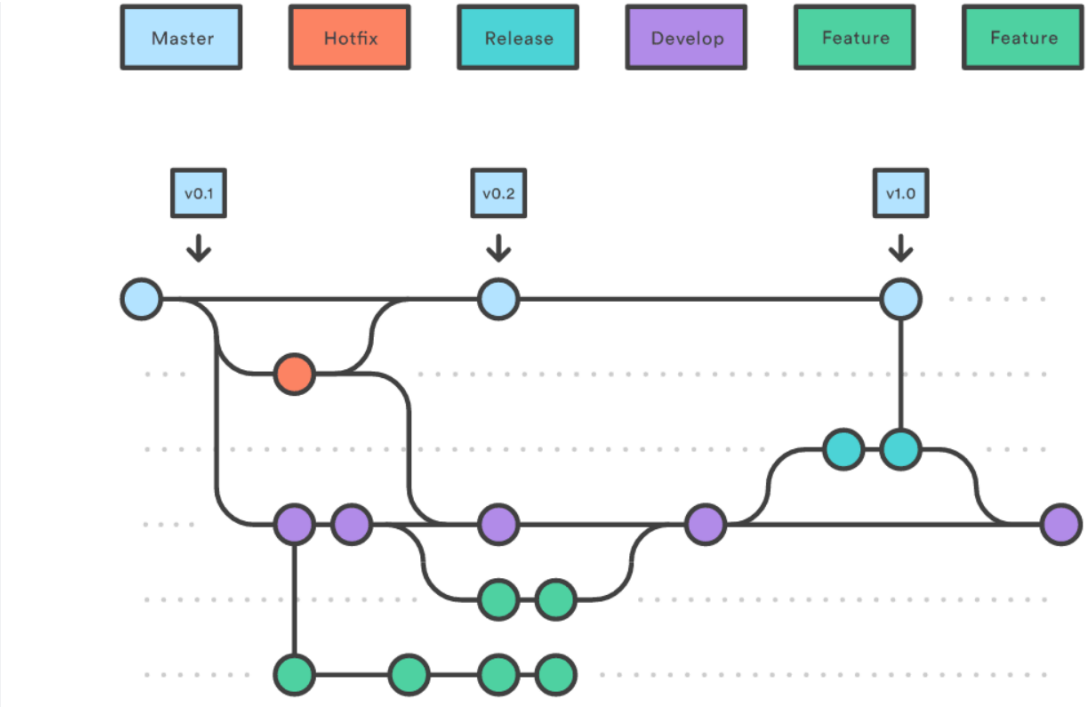
A Forking Workflow legfőbb előnye, hogy a hozzájárulások (contributions) integrálhatók anélkül, hogy mindenkinek egyetlen központi repository kellene push-olni, amely tiszta projektelőzményhez vezet. A fejlesztők saját kiszolgálóoldali repository-ra lépnek, és csak a projektfenntartó léphet a hivatalos repository-ba.

Amikor a fejlesztők készen állnak egy helyi commit közzétételére, a saját nyilvános - nem a hivatalos –repository-ba pushol-ják a commit-ot. Ezután egy pull request-et nyújtanak be a fő repository-n, amely a projekt fenntartójának tudatja, hogy a frissítés készen áll az integrálásra.

Meg tudná magyarázni a Gitflow munkafolyamatot?

A Gitflow munkafolyamat két párhuzamos, régóta futó ágat alkalmaz a projekt történetének rögzítésére, a master és a develop elvégzésére:

- Master - mindig készen áll a LIVE-on való kiadásra, mindent teljesen tesztelve és jóváhagyva (gyártásra – production- kész).
- Hotfix - A karbantartási vagy a „gyorsjavítás" ágakat a gyártási (production) kiadások gyors javításához használják. A gyorsjavító ágak sokban hasonlítanak a release és a feature ágakhoz, kivéve, hogy a fejlesztés helyett a masteren alapulnak.
- Develop - az az ág, amelyhez az összes jellemző ág összeolvad, és ahol az összes tesztet elvégzik. Csak akkor lehet összevonni a masterrel, ha mindent alaposan ellenőriztünk és kijavítottunk.
- Feature - Minden új szolgáltatásnak a saját ágában kell lennie, amelyet a development ághoz lehet push-olni, mint szülőjüket.



Írja le a "Rebase Workflow" git parancsok sorozatát

Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

1. Create a feature branch

```
$ git checkout -b feature
```

2. Make changes on the feature branch

```
$ echo "Bam!" >>foo.md
$ git add foo.md
$ git commit -m 'Added awesome comment'
```

3. Fetch upstream repository

```
$ git fetch upstream
```

4. Rebase changes from feature branch onto upstream/master

```
$ git rebase upstream/master
```

5.Rebase local master onto feature branch

```
$ git checkout master
$ git rebase feature
```

6. Push local master to upstream

```
$ git push upstream master
```

Mi a "HEAD" Gitben?

A HEAD egy hivatkozás (hivatkozás) a jelenleg kijelölt commit-ra.

Normál állapotban ez valójában egy szimbolikus hivatkozás az Ön által leírt ágra - ha megnézzük a .git / HEAD tartalmát, akkor valami olyasmit látunk, hogy "ref: refs / heads / master". Maga az elágazás utal az elágazás csúcsán elkövetett commit-ra. Ezért normális állapotban a HEAD gyakorlatilag az aktuális ág csúcsán végzett commit-ra utal.

Lehetséges leválasztott (detached) HEAD is. Ez akkor történik, amikor a (helyi) ágon kívül megnéz valamit, például egy távoli branch-et, egy adott commit-ot vagy egy címkét (tag). Ennek leggyakoribb helye egy interaktív újraindítás során van, amikor egy commit szerkesztését választja. Leválasztott (detached) HEAD állapotban a HEAD közvetlen hivatkozás egy commit-ra- a .git / HEAD tartalma SHA1 hash lesz.

Hogyan lehet eltávolítani a fájlt a git-ből anélkül, hogy eltávolítanánk a fájlrendszeréből?

Ha nem vagy óvatos a git add parancs során, akkor végül olyan fájlokat adhat hozzá, amelyeket nem akartál commit-olni. A git rm azonban eltávolítja mind az átmeneti területről (index), mind a fájlrendszeréről (woking tree), ami nem biztos, hogy a kívánt.

Ehelyett használja a git reset parancsot:

```
git reset filename          # or
echo filename >> .gitingore # add it to .gitignore to avoid re-adding it
```

Ez azt jelenti, hogy a git reset <paths> ellentéte a git add <paths> -nek.

Mi a különbség a "git stash pop" és a "git stash pop" között?

A git stash pop eldobja a (alapértelmezés szerint legfelsõbb) stash-t az alkalmazást követõen, míg a git stash-re kattintva a késõbbi újrafelhasználás céljából a stash listában marad (vagy el is dobhatja).

Ez akkor történik, hacsak nem ütköznek konfliktusok a git stash pop után (mondjuk, hogy a stashed módosítások ütköznek más módosításokkal, amelyeket a stash létrehozása óta hajtott végre), ebben az esetben ez nem távolítja el a stash-t, pontosan úgy viselkedik, mint a git stash.

Másik megközelítésben: a git stash pop az alábbiak kombinációja: git stash apply && git stash drop.

Meg tudja magyarázni, hogy mit csinál a „git reset” egyszerű angol nyelven?

Általánosságban elmondható, hogy a git reset funkció az, hogy az aktuális elágazást veszi, és visszaállítja máshová, és esetleg magával hozza az indexet és a wotking tree-t.

```
- A - B - C (HEAD, master)
# after git reset B (--mixed by default)
- A - B (HEAD, master)      # - C is still here (working tree didn't change state), but there's no
branch pointing to it anymore
```

Ne feledje, hogy gitbe van:

- a HEAD mutató, amely megmondja, milyen commit-tal dolgozik
- a working tree, amely a rendszer fájljainak állapotát képviseli
- a staging are(más néven index), amely "szakaszokat (Stages)" megváltoztat, hogy később együtt lehessen őket commit-olni

- `git reset --soft` moves HEAD but doesn't touch the staging area or the working tree.
- `git reset --mixed` moves HEAD and updates the staging area, but not the working tree.
- `git reset --merge` moves HEAD, resets the staging area, and tries to move all the changes in your working tree into the new working tree.
- `git reset --hard` moves HEAD and adjusts your staging area and working tree to the new HEAD, throwing away everything.

Használati esetek:

- `--soft` amikor át akar lépni egy másik commit-ra és javítani akarja a dolgokat anélkül, hogy "elveszítené a helyét". Elég ritka, hogy szükséged lenne erre.
- `--mixed` (ami az alapértelmezett), amikor meg akarja nézni, hogy néznek ki a dolgok egy másik commit-nál, de nem akarja elveszíteni a már végrehajtott változtatásokat.
- `--merge` amikor új helyre akar menni, de a már meglévő változtatásokat beépíti a working tree-be.
- `--hard` mindent kitörölni és új lapot kezdeni az új commit-nál.

Írjon le egy git parancsot a két commit közötti különbség ellenőrzésére

A git diff lehetővé teszi az ágak vagy a commit-ok közötti különbségek ellenőrzését.

```
git diff <branch_or_commit_name> <second_branch_or_commit>
```

Hogyan lehet módosítani a régebbi Git commit-okat?

```
git rebase -i HEAD^^^
```

Most jelölje meg a módosítani kívántakat szerkesztéssel vagy e-vel (cserélje ki a jelölést). Most mentse és lépjen ki.

Most végezze el a módosításokat, majd:

```
git add -A
git commit --amend --no-edit
git rebase --continue
```

Ha további törlést szeretne hozzáadni, távolítsa el az opciókat a comm parancsból. Ha módosítani szeretné az üzenetet, hagyja ki a --no-edit opciót.

Milyen git parancsot kell használnia ahhoz, hogy megtudja, ki módosította az adott fájl bizonyos sorait?

Használja a git blame-et - a git egy kis funkciója, amely lehetővé teszi, hogy lássa, ki mit írt a repository-ba. Ha tudni szeretné, hogy ki módosított bizonyos sorokat, akkor a -L jelzővel megtudhatja, hogy ki változtatta meg ezeket a sorokat. Használhatja a következő parancsot:

```
git blame -L <line-number>,<ending-linenumber> <filename>
```

Mikor használja a "git rebase" szót a "git merge" helyett?

Mindkét parancsot úgy tervezték, hogy integrálja az egyik ágból a másikba történő változásokat - egyszerűen csak nagyon különböző módon teszik.

Fontolja meg merge/rebase előtt:

```
A <- B <- C    [master]
^
 \
  D <- E       [branch]
```

after `git merge master`:

```
A <- B <- C
^         ^
 \         \
  D <- E <- F
```

after `git rebase master`:

```
A <- B <- C <- D <- E
```

Rebase-el azt mondod, hogy használj másik ágat új base-ként a munkádhoz.

Mikor kell használni:

- Ha kétségei vannak, használja a merge-t.
- A rebase vagy merge választása az előzmények megjelenése alapján.

További megfontolandó tényezők:

- Megosztja-e azt a branch-et, amelyen változtatásokat kap a csapatán kívüli más fejlesztőkkel (pl. Nyílt forráskódú, nyilvános)? Ha igen, ne használja a rebase-t. A Rebase elpusztítja az ágat, és ezeknek a fejlesztőknek hibás / inkonzisztens repository-k lesznek, hacsak nem használják a git pull --rebase alkalmazást.
- Mennyire képzett a fejlesztő csapata? A Rebase romboló művelet. Ez azt jelenti, hogy ha nem megfelelően alkalmazza, elveszítheti az commit-olt munkát és / vagy megsértheti más fejlesztői repository-k következetességét.
- A repository maga is hasznos információt képvisel? Egyes csapatok az ágankénti elágazás modellt használják, ahol minden elágazás egy funkciót (vagy hibajavítást, vagy aljellemzőt

jelent, stb.) képvisel. Ebben a modellben az elágazás segít azonosítani a kapcsolódó commit-okat. Fejlesztői ágonkénti modell esetén maga az ág nem ad át további információt (az elkötelezett már rendelkezik a szerzővel). Nem árt a rebase.

- Bármilyen okból visszavonja a merge-t? A visszavonás az rebase-hez képest meglehetősen nehéz és / vagy lehetetlen (ha a visszaállításnak konfliktusai voltak). Ha úgy gondolja, hogy van esély visszavonni, akkor használja a merge-t.

Tudja, hogyan lehet egyszerűen visszavonni a git rebase-t?

A legegyszerűbb módja az, hogy megtalálja az ág head commit-ját, mint közvetlenül az rebase megkezdése előtt volt a reflogban

```
git reflog
```

és állítsuk vissza az aktuális elágazást (azzal a szokásos figyelmeztetéssel, hogy a --hard opcióval történő visszaállítás előtt teljesen biztosak legyünk).

Tegyük fel, hogy a régi commit HEAD @ {5} volt a ref naplóban:

```
git reset --hard HEAD@{5}
```

A rebase is elmenti a kezdőpontot az ORIG_HEAD fájlba, így ez általában olyan egyszerű, mint:

```
git reset --hard ORIG_HEAD
```