**Git és Github ismertető**

A fejlesztők körében leginkább elterjedt verziókezelő rendszer a Git. Népszerűségét elsősorban rugalmasságának köszönheti. Olyan rendszerek alapja, mint például a GitHub, ami a nyílt forrású projektek legnagyobb gyűjtőhelye. Ezt a cikket egy nagyon rövid bevezetőnek szánom, ami segít megérteni a Git működését, és azt, hogy hogyan járulhatunk hozzá nyílt forrású projektek fejlesztéséhez a GitHub-on. Ahhoz, hogy a Git működését megértsük, érdemes kicsit benézni a motorháztető alá…



A Git-et legegyszerűbben egy fájlrendszerként képzelhetjük el. Amikor létrehozunk egy verziót (commit), egyszerűen fogjuk a projekt aktuális változatát, és kimásoljuk egy könyvtárba, aminek a neve a felhasználói nevünkből és egy dátumból áll össze. Mondjuk valami ilyesmi: TheBojda\_201902242053. Aztán fejlesztgetünk tovább, és ha megint úgy gondoljuk, létrehozunk egy új verziót, vagyis megint kimásoljuk az aktuális fájlokat egy könyvtárba. A verziókat aztán visszanézhetjük, megnézhetjük mi változott, vagy visszatérhetünk egy régebbi verzióra ha szükséges. A verziókon kívül van egy másik fogalom is Git-ben, az ág (branch). Ha létrehozunk egy ágat, akkor az egész projektet verziókkal együtt kimásoljuk egy külön könyvtárba, aminek szabadon választható nevet adhatunk. A Git tehát egy olyan fájlrendszer, ami branch-eket tartalmaz (ezeket most képzeljük könyvtáraknak), amin belül commitok (verziók — ezeket is képzeljük könytáraknak a branch-eken belül) vannak. Mivel mindig valamilyen branch alatt vagyunk, ezért létezik egy alapértelmezett branch, amit master-nek hívnak.

Persze ha a Git egyszerű fájlrendszer lenne, rengeteg helyet foglalna. Ha van például egy PHP alapú oldalunk ami 10Mb helyet foglal, és csinálunk belőle 10 commitot, amiből aztán létrehozunk 10 branch-et, az 1Gb-nyi helyet foglalna el a merevlemezünkön. Szerencsére a Git nem átlagos fájlrendszer, sokkal inkább hasonlít az IPFS-re.

Az IPFS-ről nemrég írtam egy cikket, amit mindenképp érdemes elolvasni, ha valakit érdekelnek a részletek.

Dióhéjban annyi a lényeg, hogy az olyan hash alapú fájlrendszerek, mint az IPFS vagy a Git egy adott fájlt csak egyszer tárolnak, akárhány példány is legyen belőle a fájlrendszerben. Innentől kezdve nem érdekes, hogy hány verziót vagy branch-et hozunk létre, ezek szinte semmi helyet nem foglalnak, hiszen csak a megváltozott fájlokat kell pluszban eltárolnunk. A Git ráadásul tud még egy trükköt. Képes a tárolót egyetlen pack fájlba tömöríteni, ami az egyes fájl változatok közötti változásokat tárolja csak, ráadásul ezt is tömörített formában. Ennek az okos tárolásnak köszönhetően nagyon gyorsan és minimális tárhely felhasználás mellett készíthetünk branch-eket. Úgy is szokták mondani, hogy a Git esetén “olcsó” a branch-elés, amit a Git-et használó fejlesztők erősen ki is hasznának (erre később látunk majd példát). A Git a branch-eket és a verziókat a fent leírt tömörített formában a .git köyvtárban tárolja. Ha dolgozni akarunk egy változattal, azt ki kell bontanunk. Ezt hívjuk checkout-nak.

Persze az egész rendszer nem sokban segítené a közös munkát, ha csak egy lokális fájlrendszer lenne. Éppen ezért a Git lehetőséget biztosít ilyen lokális tárolók (repository-k) szinkronizálására. Ennek két módja lehet: a tároló áthúzása (pull) és áttolása (push). Maga a tároló lehet a fájlrendszer egy másik pontján, vagy egy távoli gépen, amit HTTP(S)-en vagy SSH-n érhetünk el. Ez a két legelterjedtebb metódus a távoli tárolók elérésére, de a Git flexibilis felépítésének köszönhetően más megoldások (pl. IPFS támogatás) is szóba jöhetnek.

Amikor egy távoli repository-t húzunk le (pull), a Git befésüli a commit listát a helyi tároló listájába, valamint a fájlok aktuális állapotát is megpróbálja összefésülni. Ha szerencsénk van, akkor a távoli repository-ban más fájlok változtak, mint helyben, így nem lesz probléma. Előfordulhat azonban az, hogy velünk egy időben egy másik fejelsztő is ugyanazzal a fájlal dolgozik mint mi és a rendszer nem képes összefésülni a helyi és a távoli változatot. Ekkor ütközés (conflict) keletkezik, és létrejön egy olyan fájl, ami a két változat összefésülésének eredménye, és meg vannak benne jelölve a problémás részek. Ezeket csak a fejesztő tudja rendbe rakni kézzel, hiszen ehhez érteni kell a kódot. Ha minden conflict feloldásra került, létre kell hozni egy új commitot, ami már a korrekten összefésült (merge-ölt) változatot tartalmazza.

A fordított irányú szinkronizálás, vagyis a változások áttolása (push) jóval egyszerűbb, hiszen push-olni csak akkor tudunk, ha előtte lehúztuk (pull) a távoli tároló tartalmát és mindent rendberaktunk. A távoli tárolókat remote-nak nevezzük, és akárhány lehet belőlük. Hivatkozhatunk rájuk URL-el, vagy el is nevezhetjük őket. Azt is megtehetjük, hogy a lokális tárolónkat egy távoli tároló másolásával (clone) hozzuk létre. Ekkor a távoli tároló tartalma letöltődik a helyi gépünkre, és a fő ág (master branch) legutolsó változata lesz kibontva (checkout) a lokális fájlrendszerbe. Ilyen esetben létrjön egy origin nevű remote is, ami a forrás URL-re mutat. Ez lesz az alapértelmezett célja a pull/push műveleteknek, ha nem adunk meg más remote-ot.

Most, hogy már szinte mindent tudunk a Git-ről, pár példán keresztül nézzük meg, hogy miként néz ki mindez a gyakorlatban. Arra fogok rövid példát mutatni, hogy hogyan járuhatunk hozzá egy GitHub projekt fejesztéséhez, jelen esetben az ENVIENTA Platformhoz. A GitHub logikája szerint ha szeretnénk beszállni egy projekt fejelsztésébe, ahhoz előszőr le kell másolni (fork-olni kell) a projektet. Olyan ez mint ha klónoznánk azt a saját felhasználónk alá, de nyilván a GitHub a Git-től ellesett okos tárolást alkalmazza, így a háttérben nem másolódik semmi. Ebből a szempontból a fork olyan mint a branch, csak eggyel magasabb szintű. A lényeg mindebből, hogy a fork után lesz egy saját példányunk, ahol kedvünkre garázdálkodhatunk, hiszen az csakis a miénk. Ahhoz, hogy valami érdemlegeset tudjunk kezdeni a repository-val, le kell azt klónoznunk a saját gépünkre. Ezt a következő git paranccsal tehetjük meg:

git clone https://github.com/TheBojda/EnvientaPlatform.git

Itt persze az URL-t mindenki cserélje le a saját Git tárolójának URL-jére, amit GitHub-on a Clone or download részben találhatunk.

Egy projekthez kéféle módon járulhatunk hozzá. Vagy keresünk egy feladatot (issue) magunknak, vagy mi találunk ki valami új funkciót, esetleg jelzünk valami hibát, amit egyből javítanánk is. Bárhogy is legyen és bármit is akarnánk csinálni, mindenképp legyen hozzá issue rögzítve a projektnél, tárgyaljuk meg a projekt fejlesztőivel a részleteket, és várjuk meg amíg hozzánk rendelik az issue-t. Ha nem így teszünk, könnyen megeshet, hogy nem fogják befésülni a módosításainkat, vagy kiderül, hogy velünk párhuzamosan más is dolgozott az adott issue-n, így a sok befektetett munka könnyen pocsékba mehet. Viszont ha megkaptuk az issue-t, indulhat a munka. Ehhez előszőr hozzunk létre egy branch-et az issue-hoz (ahogy a cikk elején említettem a Git-et használók szeretnek branch-elgetni). A branch neve akármi lehet. Érdemes valami beszédes nevet választani, és érdemes a névben feltüntetni az issue számát is. Új branch így hozható létre:

git checkout -b issue-12

A fenti git parancs létrehozza az issue-12 nevű branch-et, és checkout-olja is azt, tehát ez a branch lesz aktuálisan kibontva a fájlrendszerbe és minden commit ebbe a branch-be fog menni.

A fejelsztés során bármikor commitolhatunk a git commit paranccsal, de erre kényelmesebb valamilyen grafikus Git klienst használni, illetve a legtöbb IDE is képes kezelni a Git-et. Ha készvagyunk a fejelsztésekkel, toljuk vissza (push) a GitHub-on lévő repository-ba a változásokat.

git push

Miután felkerült a branch a GitHub-ra, létrehozhatunk egy Pull request-et. A Pull request tulajdonképpen egy üzenet a projekt felé, hogy elvégeztük a feladatot, a kód itt van a mi repository-nkban, és kérjük, hogy húzzák azt át és fésüljék bele a projekt repository-ba. Ha elküldtük a Pull request-et, a fejelsztők át tudják nézni azt, és vagy befésülik a projekt repository-ba, vagy visszadobják, hogy javítsunk még rajta. Ha javítani kellene, akkor újra checkout-olni tudjuk a branch-et, és dolgozhatunk rajta tovább.

git checkout issue-12

Azért jó, hogy minden issue-nak külön branch-e van, mert így akár több feladaton is dolgozhatunk egyszerre. Megeshet például, hogy már elkezdtünk egy új issue-n dolgozni, mikor a projekt egy fejelsztője át tudja nézni az előző kódunkat és visszaküldi nekünk javításra. Miután megvagyunk a módosításokkal, nincs más dolgunk, mint push-olni, és a Pull request automatikusan frissül. Ha a Pull request-et elfogadták, és nincs már szükség az adott issue-hoz kötött feature branch-re, akkor azt a következő paranccsal törölhetjük:

git branch -d issue-12

Persze rajtunk kívűl mások is dolgozhatnak a projekten, ami folyamatosan fejlődik a Pull request-ek befésülésével. A GitHub nem tudja automatikusan frissíteni a fork-unkat (lehetnek ütözések), így azt nekünk kell megtennünk manuálisan. Ehhez érdemes upstream néven hozzáadni egy remote-ot, ami a projekt repository-ra mutat.

git remote add upstream https://github.com/EnvientaGit/EnvientaPlatform.git

Innen már könnyen megoldható a szinkronizálás egy pull-al, ami lehúzza az aktuális változatot az upstream-ről, és egy push-al, ami kitolja azt a mi saját bejáratú fork-olt repository-nkba.

git pull upstream master

git push

Körülbelül ennyi lett volna a Git/GitHub gyorstalpaló. Mivel a Git csak az eszközkészletet adja, ezért a fent bemutatott fejlesztési folyamat csak a legegyszerűbb megvalósítás, ami tetszés szerint tovább bonyolítható. A projekt például fenntarthat magának egy development ágat, ahol a fejelsztések gyűlnek, majd bizonyos időközönként létrehozhatnak ezekből egy release-t (itt van erről egy igen részletes leírás: https://nvie.com/posts/a-successful-git-branching-model/). Olyan megoldás is elképzelhető, ahol egy nagy projektnél több távoli repository is van. Itt példul a programozók egy csoportját egy vezető programozó fogja össze, és csak ő push-ol tovább a központi repository-ba. Branch-ek és repository-k kusza hálózatán futhatnak végig a fejelsztések, míg végül összeáll a kész szoftver (vagy bármilyen tartalom, akár könyv, stb.), a döntés a fejelsztőkre van bízva.

A fentiekből remélem jól látszik a Git flexibilitása, és az, hogy miért vált uralkodó verziókezelő rendszerré viszonylag rövid időn belül. Azt is remélem, hogy ebből az írásból sokaknak sikerült megérteniük a rendszer működését és hasznát veszik majd az itt tanultaknak. Tehát kalandra fel! Segítsetek minél több open source projektnek, ezzel is kicsit jobb hellyé téve a világot…