**GIT**

* Git nedir Github nedir ??
* Git bir acik kaynak Version Control sistemidir ama biz dosya takip sistemi(DTS) olarak adlandiralim. Yani bu ne demektir?
* Github, git Uygulamasina ev sahipligi yani hostluk yapan uzakta internet üzerinde bulunan bir remote repodur. Github bir internet portalidir. Burada developerlar kodlarini sergileyebilirler.
* Github ayni zamanda bir git server’idir.
* Git deki repomuza lokal repo deriz, Githubdaki repomuza uzak repo anlaminda remote repo deriz.
* Peki version ne demektir ve biz Git’e neden ihtiyac duyariz??
* Bizim üzerinde calistigimiz dosyamizin yada dosyalarimizin, üzerinde degisiklik yapilmadan önceki hali bir versiyondur, degisiklik yapildiktan sonraki hali ise ayri bir versiyondur. Yani türkce olarak ifade edecek olursak dosyanin yeni hali ve eski hali.
* Ve dikkat edilmesi gereken husus sudur: Bizim dosya üzerinde degisiklikler yaparak ortaya cikardigimiz sonuc, yeni bir versiyon üretmektir.
* Somut bir örnek verecek olursak, dosyamizin eski hali IOS 11, yeni hali ise IOS 12 diyebiliriz. Iste version control sistemi ismi buradan gelmektedir. Git uygulamasi bizim dosyalarimizin en eski hallerini de en güncel hallerini de hafizasinda tutarak, bizleri ekstradan klasörleme yada arsivleme zahmetine sokmadan bizim tüm calismalarimizi bizim adimiza takip eder ve kaydeder. Ve hafizasinda tuttugu bu kayitlar vesilesi ile bizler istedigimiz zaman Git kayitlari arasinda gezinebilir ve dosyamizin ihtiyac duydugumuz tarihteki durumu ne ise o haline rahatca ulasabiliriz. Git in bize sagladigi bu özellige biz time travel diyoruz yani zamanda yolculuk.
* Düşünün bir yazılım takımınız var ve bir projeye başladınız. Projeyi modüllere ayırdınız ve her kişiye ilgili görevi atandı. Takım halinde çalışabilmeniz için bir yönetim aracı gerekli ve bir kitabın baskıları gibi bir baskı da hata çıktığı zaman hemen versiyonu değiştirip eski haline getirmeniz lazım. Burada imdadınıza (birçok iş ilanında da gördüğünüz üzere) [git](https://git-scm.com/downloads) koşuyor. En büyük avantajlarından birisi ise merkezi yapıda bulunmamasıdır. (Dağıtık Sistem)
* Dağıtık yapıda bulunur dedik. Peki bu ne demektir?
* Distributed demek, herkesin katki yapabildigi merkezi olmayan bir sistem demektir.
* Normal bir internet cafe düsündügümüzde, burada bulunan tüm pc ler ana server a baglidir. Ve ana server bir dosya üzerinde calisirken baska bir pc nin ayni dosya üzerinde calismasi olanaksizdir. Iste burada git’in bize sagladigi fayda ortaya cikiyor. Ayni salon icerisinde bulunan bilgisayarlar ayni dosya üzerinde calisamazken, biz baska baska ülkelerde olsak da git vesilesi ile bir takim calismasi yürütebiliyoruz ve farkli cografya da bulunan insanlar ile ayni proje üzerinde calisabiliyoruz.
* Git ile versiyonladığınız proje dosyaları her geliştiricinin yani takim personelinin lokal bilgisayarinda ayri ayri saklanır. Ve proje de bir değişiklik yapıldığı zaman sadece değişen kısım değil, o an projeden bir snapshot alınır (projeyi bir bütün halinde saklar) ve böylelikle yapılan katkı bir bütün halinde kolaylıkla görülmüş olur.
* Yani, ana server da yada remote(uzak) repo da herhangi bir sikinti yasandiginda diger takim üyelerinden herhangi birisi, tekrardan server a dosyayi kaybeden arkadaslarina paylasim yapabilir.
* Git in bize sagladigi bir diger fayda ise, sinirsiz sayida undo ve redo yapabilmemizdir.
* Peki undo ve redo ne demektir. Mesela Microsoft Word de bir yazi hazirliyoruz ve o esnada hata yaptigimizi fark ettik ve ctrl + Z ile islemi geri aldik. Iste bu undo’dur. Yani basi u harfi ile basladigindan u dönüsünden aklimizda kalabilir.
* Birde redo vardir. Redo ise replay dan aklimizda kalabilir. Yani ingilizce de replay tekrar oynat manasinda kullanilir, yani “re” prefix i tekrar anlami katar. Burada da hata yaptigimizi degerlendirip undo ile geri aldigimiz islemin aslinda hatali olmadigini farkedip tekrar eski haline almamiz bir redo dur.
* Git tarafindan saglanan diger bir önemli hizmet ise, Compare ve blame dedigimiz husustur. Yani git sayesinde bizler ayni dosya üzerinde farkli farkli kisiler tarafindan yapilan degisiklikleri teker teker kiyaslayabilir ve olumsuz bir durum var ise bu degisikligi yazan kisiyi yada bunun sorumlusunu tespit edebiliriz. Bu islemi gerceklestirebilmek icin git bizden islem yapmaya baslamadan önce kullanici ismimizi ve E-Mail adresimizi istemektedir.
* Compare asamasinda Git su 4 soruya cevap arar:
* 1: What changed
* 2: When it changed
* 3: Why it changed
* 4: Who it changed
* Hatirlamakta fayda var. Acik kaynak ücretsiz demek degildir. Mesela Linux’un bir distro su olan Red Hat isletim sistemi acik kaynaktir ama ücretlidir.
* Iki türlü version control system vardir:
* 1: Centralized
* 2: Distributed
* Birinci tür Vcs’ de server sürekli saglam ve ayakta olmak zorundadir. Tüm pc ler bu server a bagli olarak ve server üzerinde bulunan dosya üzerinde islem yaptigi icin serverin cökmesi durumunda üzerinde calisilan dosya tüm userlar tarafindan kaybedilir.
* Yine bu sistemde tüm kullanicilar, kesintisiz olarak internete yada baska bir aga bagli olmak zorundadir. Aksi halde takim calismasinda kopmus olacaklardir.
* Bu nedenle bu sistem takim calismasina distributed sistem kadar uygun degildir.
* Distributed olarak adlandirdigimiz sistemde ise, tüm takim calisanlari, server da bulunan dosyayi kendi bilgisayarina indirir ve herhangi bir aga bagli olmasa da cevrimdisi olarak lokalinde bu dosya üzerinde calisabilir. Ve yapacagi islemler bittiginde bir ag vasitasi ile baglanarak calismasini server a yani uzak depoya gönderir. Yani aslinda burada dikkat etmemiz gereken husus sudur:
* Bu durumda her bir takim üyesinin bilgisayari bir **REPO** olmus olur.
* Mülakat da söyle bir soru cikarsa :
* Why do we need Git?
* 1: Backup, Archieve, Versioning, History
* 2: Undo redo
* 3: Comparing(iki degisiklik arasinda farki görme)
* 4: Collaboration and Teamwork
* 5: Code review(gözden gecirme, inceleme)
* 6: Blame(sorumlu kisiyi bulma)
* Git CMD Windows komutlarini calistirir, Git Bash ise Linux komutlarini.
* **REPOSITORY:** Git uygulamasinda Repo dedigimiz kavram bir klasördür yada bir depodur. Bu klasör yada depo icerisinde bizim dosyalarimiz ve projelerimiz bulunur. Ve bu klasör **.git** adinda bir gizli klasörden ibarettir.
* Eger repomuzu silmek istersek .git isimli gizli klasörü silmemiz yeterli olacaktir.
* Dikkat: Bu bir gizli dosya oldugu icin, bu dosyayi görebilmemiz .git komutu ile mümkündür.
* Örnek: Git uygulamasinda kesintisiz bir video kaydi yoktur. Yani bizim calismalarimiz git tarafindan sürekli olarak ve otomatik olarak **kayit altina alinmaz**. Biz ne zaman kayit altina almak istersek o zaman dosyanin anlik bir fotografini cekeriz ve bunu arsive kaydederiz. Sonra bir degisiklik yaptigimizda bir fotograf daha cekeriz. Ve time travel dedigimiz islem, aslinda gecmiste cekmis oldugumuz aile fotograflarina dönüp bakmak gibidir.
* Local ve remote olmak üzere iki repomuz vardir. remote repoya ayni zamanda central repo da denilir.
* **GIT DE KULLANDIGIMIZ KOMUTLAR:**
* 1: git --version: bilgisayarimizda yüklü olan git versionunu gösterir. Not: Ubuntu da farklidir git bash de farklidir.
* 2: git config --global user.name “yourname” (Not: username yada useremail yazdiktan sonra tirnak acmadan önce bir bosluk birakmaz isek kodumuz calismaz)
* Not: Burada global ne demektir ??
* 3: git config --global user.email “youremailaddress”
* 4: git config --global core.editor “vim / Nano / Not Defteri/ ……. “
* 5: git config --list
* Biz buraya vim yada baska bir editor ismi yazmazsak o kendisi default olarak bir text editör acar.
* Not: eger isim yada email adresimizi yanlis girersek ayni komutu bir daha girdigimizde eskisini siler yenisini üzerine yazar.
* Not: Su ana kadar yaptigimiz kayit islemlerini bir kez yapmamiz yeterlidir. Ancak Git bash ve Ubuntu farkli mecralar oldugu icin her ikisinde de ayri ayri yapmak gerekir.
* Ayrica hep ayni pc den baglaniyorsak problem olmaz .Ama isletim sistemi degisirse o zaman yeniden kayit islemi yapmamiz gerekir.
* 6: explorer.exe : Not: explorer.exe komutu ile explorer.exe . komutu arasindaki farki biliyor muyuz??
* **REPO ILE ILGILI KOMUTLAR:**
* 1: git init:(initialize) Bir klasör altinda dosya takibini baslatmak icin kullaniriz.

DIKKAT: Hangi dosyamizi takip ettirmek istiyorsak onun altinda baslatiriz. Yine de yanlis yerde baslatti isek. .git isimli gizli klasörü sileriz ve takibi durdururuz.

Not: Bu komuttan sonra ubuntu da branch ismi cikmaz ama git bash de cikar.

Not: Eger bizim dosyalarimiz Github gibi uzak repolarda da mevcut ise; biz kendi bilgisayarimizda .git klasörünü sildigimizde github daki bilgiler son hali ile kalir. Cünkü orasi uzak bir repodur.

\*\*\*\* Interview Question: git de yeni bir repository hangi komut ile olusturulur ????

* 2: git help : Bu komut ile git ile ilgili komutlar hakkinda yardim alabiliriz.
* 3: git help -a: 'git help -a' and 'git help -g' list available subcommands
* Not: Bu help bölümlerinden q ile cikis yapariz.
* 4: git help -g:
* 5: git help commandname: specific olarak herhangi bir komut hakkinda bilgi almak istersek o zaman git help komutu yanina komut ismi yazilir.
* Not: Bu komutlarin Linux de nasil oldugunu kisaca hatirlayalim:
* man commandname(mesela man pwd yazdigimizda pwd komutu ile ilgili man dosyasina ulasiriz. Cikis islemleri yine q ile)
* info commandname
* help commandname
* **6: git status:**
* icerisinde bulundugumuz klasörün yada diger klasörlerin git tarafindan takip edilip edilmedigini kontrol etmenin birkac yöntemi vardir:
* Bunlardan bir tanesi Git bash kullanicisi isek bulundugumuzun klasörün yaninda branch ismi yaziyor yada yazmiyor olmasi durumudur.
* Ikincisi ise git status komutunu calistirarak dosyamizin güncel durumunu görmektir.
* Ücüncü yöntem sudur: icerisinde bulundugumuz klasör icerisinde ls -al ls -a komutunu calistirarak .git isimli gizli klasör olup olmadigini kontrol etmektir.
* Aklimiza gelen baska yöntem var midir?
* **git status :** Bu komut bizim yaptigimiz degisiklikler ile commitler arasinda kiyaslama yapar ve neyin kaydedilip neyin kaydedilmedigini bize söyler. Hatta uzaktaki repo da baska biri tarafindan bir degisiklik yapildi ise o durumda da bize der ki; sizin reponuz bir commit geride yada ileride.
* **7: git add . yada “git add dosya adi”:**
* Bu komuta gelmeden önce git icerisinde bulunan 3 asamayi inceleyelim:
* **1: Working directory** (ciftlerin yeni tanisma zamanlaridir. Bu asamada birbirlerini tanimaya calisirlar ve henüz bir karar verilmemistir.)
* **2: Staging Area** (Nisanlilik dönemidir. Ara bir karar verilmistir. Esler birbirleri hakkinda; bu kisi benim esim olabilir diyerek tanistigi bu kisiyi es adayi kategorisine almislardir. Ancak henüz kesin bir karar verilmemistir ve evlilik öncesi bir düsünme asamasidir. Eger bir problem yasanacak olursa, geri dönüs ihtimali vardir. )
* **3: Repository:** (Karsimizdaki kisi hakkinda kesin karar verilmis ve es olarak kalici hale getirilmistir. Artik bu asamadan sonra zaman da yolculuk vardir. Yani ihtiyac oldugunda dügün öncesinde ve dügün asamasindaki fotograflara bakilarak, o zaman diliminde ne durumda oldugumuz görülebilir. Is bu asamaya geldikten sonra tekrar yeni tanisma asamasina dönme sansimiz yoktur. Sadece tek ihtimal vardir. O da evliligin sonlanmasi dir yani bu da **.git** klasörünün silinmesi ile tüm takibin sonlandirilmasidir.)
* git init ile baslattigimiz dosya takibinde su ana kadar dosyamiz untracked durumdadir.
* yani türkce olarak aslinda dosyamiz su an git tarafindan izlenmiyor demektir. Biz su ana kadar sadece git in marsina bastik ve git su an bizim klasörümüz altinda rölanti de calisiyor. Yani Bizim git i harekete gecirmemiz icin öncelikle vitesi 1 konumuna almamiz ve biraz da gaza basmamiz gerekir. Bu islem de git add komutu ile gerceklestirilir.
* **NOT:** git le takip edilen bir dosyanin adini degistirirsek git eski isimli dosyayi adi degismis olarak degil deleted olarak algilar. Burada önemli husus dosyanin sadece ismini degistirsek de git add yapmak zorundayiz. Yoksa bu degisiklik takip edilmez.
* **Not:**  Staged asamasindaki bir dosyayi silersek ve tekrar dosya versiyonlari arasinda checkout yapsak da o dosya geri gelmez. Cünkü o dosya commit yapilarak hic kayda gecmemistir. Yani hicbir islem yapilmamis gibi olur.
* **\*\*\*\*\*\*Soru:** staging Area ya alinan bir dosya kalici degil gecici bir hafizaya alinmistir. Bu nedenle belirli bir süre sonra silinir. Ancak ubuntuyu acip kapattigimizda kontrol ettigimize silinmemis oluyor. Peki staging area daki bu dosya ne zaman silinir????
* 8: **git commit -m :** son olarak kesin karar verdigimiz komuttur.
* Not: Dosyalarimizi önce staging area ya sonra repoya atmak zorunda degiliz. Elimizdeki diger secenek ise. dogrudan working directory den repoya atmaktir. Bu islem git **commit -am “mesajimiz”** yada git **commit -a -m “mesajimiz”** komutlari ile gerceklestirilir.
* NOT: Mesela üzerinde calistigimiz 100 tane dosyamiz oldugunu varsayalim ve biz bunlarin sadece 20 tanesinindeki degisiklikleri commit etmek istiyoruz. o zaman sadece 20 tanesini biz staging area ya atariz. Yani secme islemini staging area ya atmadan önce yapariz.
* Biz git init yapmazsak git degisikligi fark eder ancak takip etmez.
* **git restore ilhan.txt son degisikligi geri alir**
* **git log --oneline**
* EGER MÜLAKATTA BU TARZ BIR SORU CIKARSA
* Soru: Lütfen git deki workflow’u aciklayiniz:
* 1: Working Directory: Where you work. Create new files, edit files, delete files etc.
* NOT: explorer.exe . komutunu calistirdigimizda , bizim karsimiza cikan .git klasörü haricindeki dosya ve klasörler bizim working directory mizdir. .git isimli klasör ise bizim git uygulamasindaki yani lokaldeki repomuzdur. Biz uygulamis oldugumuz tüm undo ve redo islemlerini .git isimli klasör icerisinde tutulan bilgiler vasitasi ile yapariz.
* Not: Bu arada explorer.exe komutu gercek ubuntu isletim sisteminde calismaz. Sadece git bash ve wsl olarak kullanilan ubuntu da calisir.
* 2: Staging Area: Before taking a snapshot, you are taking the files a stage. Ready files to committed.
* 3: Repository: Committed snapshots of your project will be stored here with a full version history.

**FILE STAGES**

* \*\*\*\*\*\* Dikkat. File stages i workflow ile karistirmayalim. 3 adet file stages vardir. Aslinda workflow ile dogrudan iliskilidir ama ayni sey degildir:
* 1: modified: changes made to files since last commit snapshot. Bu asamada git status komutu calistirildiginda dosya renginin kirmizi oldugu ve yaninda untracked yazdigi görülür.
* 2: staged: changes marked to be added into the next commit snapshot. Bu asamada dosya rengi artik yesile dönmüstür. ve dosya ismi üzerinde su yazi görülür:
* Changes to be committed:
* (use "git rm --cached <file>..." to unstage)
* 3: committed: unmodified changes from the last commit snapshot. Bu asamada dosyamiz kalici repoya aktarilmistir. git commit -m komutundan sonra su yazi görülür:
* [master (root-commit) ea3f674] degisiklik yapildi
* 1 file changed, 0 insertions(+), 0 deletions(-)
* create mode 100644 test.txt
* **git log:** diger bir komutumuz ise git log dur. Bu komut yazdigimizda karsimiza cikacak hususlar sunlardir:
* 1: degisikligi yapan kisinin kullanici adi ve email adresi
* 2: degisiklik yapilan tarih saat dakika ve saniye
* 3: zamanda yolculuk yapmak icin ihtiyac duydugumuz commit numarasi
* 4: commit mesaji
* 5: son olarak da HEAD simgesinin nerede bulundugu ve hangi branch üzerinde bulundugumuz (\*\*\*\*\*\*\* Önemli: git log komutundan sonra HEAD simgesi hangi commit üzerinde bulunuyorsa, biz an itibari ile dosyanin o hali üzerinde bulunuyoruz demektir. Ve bu ayni zamanda bize üzerinde bulundugumuz branch i gösterir. Bu durumu yaparak test edelim )
* Öncelikle su an git log yazarak güncel duruma bakalim.
* sonrasinda hem yeni bir branch olusturup hem de o branche gecelim
* ve dosya üzerinde bir degisiklik yaparak commit edip git log yapalim.
* Bu komut sonrasinda HEAD simgesinin yeni branch yaninda oldugunu ve her ikisinin de en son commit yaninda oldugunu görecegizdir.
* **git log --since=15minutes**
* **git add . komutuna alternatif komutlar:**
* **git add -A**  (stages all changes: yapilan tüm degisiklikleri, yani sayisi birden cok da olsa hepsini staging area ya gönderir yani git add . gibi calisir.
* Not: git add komutu su sekilde calismaz.
* **git add -a** yada **git add -al** ⇒> bunlar calismaz
* a parametresi ile kullanmak istiyorsak **A** ile kullanmamiz gerekir.)
* **git add -u** (stage modified and deleted files only : Bu kodun calisma prensibini biliyor muyuz?? degisen dosyalari stage etmiyor. Silinen dosyalari denedim onlari da etmiyor.)
* \*\*\*\* Önemli Not: Yeni olusturdugumuz bir dosyayi git commit -am komutu ile, working directory den dogrudan repo ya atamayiz. Her yeni acilan dosya ilk etap da git add komutu ile takibe alinmak zorundadir. Ancak bundan sonraki asamalarda dosyada degisiklik olursa o zaman git commit -am kullanilabilir.
* **git commend --amend:**
* Bu komut ile son yaptigimiz commit’in commit mesajini degistirebiliriz. Calisma prensibi su sekildedir. git commit --amend yazdigimizda, ilk etapda git üzerinde kayit yaparsak git config --global core.editor komutunda hangi editörü yazdi isek o editör acilir ve bu sekilde istedigimiz degisikligi yapar kaydede cikariz.
* NOT: Burada dikkatimizi cekmesi gereken husus: --amend komutunda iki tane -- bulunuyor olmasidir. Eger bir tane yazarsak kodumuz calismaz.
* Sonrasinda yaptigimiz bu degisikligi yine git log komutu ile kontrol ederiz.
* **git rm --cached dosyaadi:**
* Bu komut ile staging area ya gönderdigimiz dosyalari tekrar working directory e geri cekeriz, yani rengi yesile dönüsen dosyamizin rengi tekrar kirmizi olur. Burada dikkat etmemiz gereken husus dosya adini yazmayi unutmamaktir. Yani mesela birden cok dosya göndersekte tek dosya göndersekte sadece git rm --cached yazdigimizda kodumuz calismaz.
* **git restore --staged dosyaadi** : Bu koda Dikkat edersek ingilizce olarak staged edilmis alani restore etmemizi yani onarmamizi tadilat etmemizi yaziyor. Yine bu komutta da dosya adi yazilmaz ise kodumuz calismaz.
* Sizler icin denedim. **Bu iki komutta da “-a -al veya -A” parametreleri calismiyor.**
* Birden cok dosyayi tek satir komut ile staging area dan tekrar working directory ye almanin yolu su sekildedir:
* yine bu iki komuttan birisi yazilir ve sonrasinda bir bosluk birakilarak dosya isimleri yazilir. Ve her bir dosya adi arasinda bir space birakilir. Burada istersek dosya isimlerini “ ” icerisinde yazabiliriz, istersek “ ” kullanmayabiliriz. Iki sekilde de kodumuz calisir.
* **git checkout commitnumarasi**: Bu komut ile dosyalarimizi commit ettikten sonra, zamanda yolculuk yapabilir ve dosyalarimizin versiyonlari arasinda gezinebiliriz.
* NOT: git checkout commitno komutu ile eski bir commitimize yolculuk yaptigimizda artik, git log komutuyla en son yapmis oldugumuz commitleri göremeyiz. Bu durumda su komut kullanilir:
* **git checkout master** (eger yine göremezsek degisiklikleri kendisi üzerinde gerceklestirdigimiz branch’ e gecilir.)

**BRANCH VE HEAD**

* Head: Bir pointerder. Bizim hangi commit üzerinde oldugumuzu gösterir. Head in simgesi \* dir. \* nerede ise Head oradadir.
* Branch: Türkce karsiligi dal veya kol demektir. Ve bize, ana branchimizde ve dosyalarimizda hicbir problem olmadan takim halinde calisma olanagi sunar. Bir soyagaci olarak düsünebiliriz. En üstte bir büyükbaba ve sirayla dagilan aile yapisi.
* Ana branchlarimiz main yada master adi ile belirtilir. Eger istersek biz bu isimleri de degistirebiliriz. Ama sirketler de ve projelerde ana branchlar master yada main olarak kullanilir.
* Ana branchimizi degistirmek icin kullanacagimiz komut su sekildedir:
* git branch -M branchname (Bu komut ile yeni bir branch olusturup tekrar o branch a gecmeye ihtiyac duymayiz. Bu komut ile dogrudan üzerinde bulundugumuz branchin adini degistiririz. )
* **Not:** Github ise artik Irkcilikla ilgili bir sebepten dolayi “master” branch ismi olarak kullanilmamaktadir.
* **Branch yapisinin olusturulma sebebi:**
* Kendimiz evimizde yada is yerinde kimsenin dahil olmadigi bir dosya yada projede calisirken hicbir sikinti yoktur. Istedigimiz degisikligi yapar istedigimiz sekilde kayit yapariz. Ama bir dosya üzerinde 100 kisinin calistigi bir durum düsünelim. Gün icerisinde yüzlerce commit yüzlerce degisiklik arasinda catisma cikmamasi sikinti yasanmamasi imkansiz gibidir. Iste bu sikintiyi ortadan kaldirmak icin branch yapisi olusturulmus ve herkesin gerekirse bir brancha ayrilarak yine kendi evinde calisiyormus gibi rahat bir sekilde degisiklikler yaptigini ve bunu istedigi gibi kaydettigini düsünebiliriz.
* Söyle düsünebiliriz. Bir devlet kurumu. Bir genel müdür ve onlarca daire baskani var. Her bir daire baskani kendisi ve astlari ile kendi dairesi icerisinde bir calisma yürütüyor. Bu kisi kendi dairesi sinirlari icinde istedigi gibi calismalar yapiyor. ve bu calismalardan genel müdürlügün hicbir haberi olmuyor ve bir hata yapilirsa da henüz bu calismalar faaliyete gecmedigi icin genel müdürlügü kesinlikte etkilemiyor. Ta ki ne zamana kadar. Daire baskani kendi dairesinde islerini olgunlastirip, projesini genel müdüre onaya cikarana kadar. Iste bu asamada proje onaya cikiyor. Ve genel müdür isterse bu projeyi onayliyor ve genel müdürlügün yönetimine dahil ediyor.
* **Interview:** Branch yapisinin bizlere sagladigi en büyük fayda nedir?. Herkes kendi sinirlari icerisinde istedigi degisikligi yapabilir ve diger personeller bundan asla etkilenmez. Yani en önemli özelligi bizlere Takim calismasi imkani sunmasidir.
* Branch sisteminin bize sagladigi en önemli ikinci fayda sudur:
* Hicbir sahis yada sirket düzgün bir sekilde calisan bir sistemin bozulmasini istemez.
* Iste bu nedenle main yada master gibi sistemin bel kemigi diyecegimiz ana branslar üzerinde hatali islem yapilmasini ve bunun commit ile kayda gecirilmesini kimse istemez. Yani demek oluyor ki. Sistemin en stabil calistigi yer master yada main branch leridir. Iste bu nedenle branch sistemi olusturulmustur. Alt branclarda yapilan hicbir degisiklikte üst kademedeki bir sahsin onayini almadan ana artere yani ana branch e aktarilmaz.
* Branch hangi ana koldan olusturuldu ise o ana branch teki dosyanin son hali ile aynidir. Yani ana branch te yeni branch olusturuldugu esnada 4 adet dosya varsa yeni acilan branch te de 4 dosya olur. Ama ayrilma gerceklestikten sonra alt branch üzerinde acilan 5. bir dosya merge edilene kadar ana dosyada görünmez.
* **Not:** Alt branch lerde yapilan hicbir commit test edilmeden ana branch a merge edilmez ve test edilen commitler artik ana branch a aktarilmak icin hazir hale gelir.
* Sonrasinda da team leader yada project manager test ettikten sonra eger problem görmezse degisiklikleri ana branch e merge eder.
* Böylece iki dal tek bir dal haline gelir ve bizim kendi dalimizda yazdigimiz tüm kodlar ana dala aktarilmis olur.
* Eger kendimiz tek calisiyorsak o durumda merge islemini kendimiz gerceklestiririz.
* Git sisteminin calismasi icin Github ile calismak zorunda degiliz. Sadece git sistemi ile lokal bilgisayarimiz da da calisabiliriz.

**BRANCH KOMUTLARI**

* **git branch** : var olan branch leri görüntüler
* **git branch -r** : remote da yani uzak depo da (github da gitlab de)bulunan branchler
* görüntülenir.
* **git branch -a** yada **git branch -al** komutu ile hem uzaktaki hem lokaldeki branchler görüntülenir
* **yeni branch olusturma**: git branch newbranchname
* branch degistirme: git checkout branchname :
* branch degisikligi yapildiktan sonra hangi branch e gecildi ise onun rengi yesil olur ve basinda yildiz olur.
* ayni anda yeni bir branch olusturma ve hem de direkt olarak o branch a gecme:

**git checkout -b newbranchname**

* Sirketlerde profesyonel hayatta genelde ana branch altinda dev yada development branch i olusturulur. Ve bu dev branch i altinda da cok sayida insan calistigi icin yine bu branch altinda da farkli branchler olusturulur.

GÖRSEL OLARAK BRANCH UYGULAMA SITESI: <https://learngitbranching.js.org/>

(siteye girince sirasiyla; yesil buton, 3 ve kirmizi buton)

* Bu sitede yaptigimiz komutlar:
* git commit -m “degisiklik ismi” > c2 olusur ve head oraya gider
* git commit -m “degisiklik” > c3 olusur ve head oraya gider
* git branch development > yeni branch olusturduk. (yeni branch olusturdugumuzda ana branchteki tüm dosyalar aynen bu yeni branch a da aktarilir)
* git checkout dev
* artik dev branch üzerinden commit yapmaya basliyoruz ve ayrilacagiz
* git commit -m “........”
* git commit -m ”.......”
* git checkout main diyerek tekrar main’e geliyoruz
* git branch new yeni branch olusturuyoruz
* git checkout new
* git commit -m “......” simdi new üzerinden commit yapiyoruz
* git checkout dev
* git branch computer (dev üzerinde branch olusturduk)
* git branch tomy (baska bir arkadas kendi branchini olusturdu)
* git checkout computer
* git commit -m “....” (computer üzerinde commit yapiyoruz)
* git checkout tomy
* git commit -m “.......”
* git checkout dev
* git commit -m “----” (dev branchta degisiklik)
* git checkout computer
* git commit -m “-...”
* git checkout dev
* git checkout -b ahmet (hem olusturduk hem de icine girdik)
* git commit -m “.....” (ahmet üzerinde degisiklik yapiyoruz)
* git branch computer
* git checkout -b feature (yeni bir branch olusturuyoruz)
* git commit -m “......”
* **\*\*\*\*\*\*\*\*\* Branch silme komutlari:**
* 1: git branch -d branchname
* 2: git branch -D branchname
* Kücük d li olan formül ile Büyük D li formül arasinda fark vardir. Eger bir bir kol üzerinden baska bir kol olusturdu isek ve bu yeni kolda degisiklikler yapti isek ve bu degisiklikler üstteki ana kola merge edilmedi ise o zaman -d ile silemeyiz git müsaade etmez.
* Ancak buna ragmen silmek istersek -D li olan komut ile silebiliriz. Yani linux deki rm -rf komutu gibi Force kullanarak siler.
* Eger biz alt daldaki degisiklikleri merge etti isek yada alt branch olusturduktan sonra yeni branch üzerinde hicbir degisiklik yapmadiysak o zaman -d ile de silebiliriz.
* **\*\*\*\*\*\*\*\*\*\* Önemli:** Eger biz bir alt dali silmek istiyorsak, ve eger o branch icerisinde isek o zaman o branch i silemeyiz. Ne yapmamiz lazim. Bir üst branch a cikip oradan alt dali silmemiz gerekir. Bu islem ayni Linux de klasör silme islemine benzer. Linux de de icinde oldugumuz klasörü silemeyiz.
* Not: Ayni komut satirinda birden cok branch ayni anda silinebilir.
* \*\*\*\* **git checkout** komutu ile hem branchler arasinda gezeriz hem de commit ler arasinda zamanda yolculuk yapariz.

**MERGE**

* **Iki türlü merge vardir:**
* Birincisi **fast forward merge**: Herhangi bir ana koldan bir branch ayrilir ve bu alt branch ayrildiktan sonra bu alt branch üzerinde commitler ve degisiklikler yapilir. Ancak bu esnada ana kol üzerinde hicbir commit yada degisiklik yapilmamis ise merge islemi esnasinda bu alt ve ana branchi birlestirmek icin yeni bir commit yapmak gerekmez. Ana kolda degisiklik olmadigi icin ve sadece alt kolda degisiklik oldugu icin merge esnasinda bu iki kol arasinda catisma cikma durumu söz konusu degildir. Ve bu nedenle artik ana branchimiz alt branch üzerinden yoluna devam eder. Eger herhangi bir zamanda alt brancha dönmek istersek, tabi ki bu islemde mümkündür.
* Asagidaki örnekte gördügümüz gibi artik ana branchimiz komple L seklinde devam etmektedir.

-----0-------------0-----------0----------I

I

I ------------------------------

* **Ikinci merge yöntemi ise 3 way mergedür**. Bu yöntemde ise durum su sekildedir:
* Yine herhangi bir ana branchten bir alt kol ayrilir ve bu alt kol ayrildiktan sonra kendisi üzerinde degisiklikler ve commitler yapilir. Ancak bu esnada ana kol üzerinde de degisiklikler ve commitler yapilir. Iste bu nedenle iki kolun birlesmesi esnasinda catisma cikma olasiligi mevcuttur. Bu nedenle de iki kol üzerinde de yapilmis olan bu degisiklikleri birbirine koordineli ve sorunsuz bir sekilde merge edebilmek icin yeni bir commit olusturulur ve alt kolda yapilan degisiklikler ana kola, ana kolda yapilan degisiklikler alt kola aktarilir. Böylece iki kol üzerinde de tüm degisiklikler mevcut olmus ve iki kol üzerindeki tüm dosyalar birbiri ile esit hale gelmis olur ve 3 yolun birlestigi yerde yeni bir commit yapilir.
* Peki 3 way ismi nereden gelmektedir??

-------------------------0---------------I-----------0-----------0----------------------

I I

I-----------0-----------I

* Dikkat: merge esnasinda hangi branch a merge etmek istiyorsak o branch e gecmemiz gerekir. ve genelde prosedür olarak alt branch üst branch e merge edilir. Bu demek olur ki merge esnasinda üst branch e gecmemiz gerekir.
* **Not:** Git de Alt branch te bulunurken üst branch e merge talebi gönderemiyoruz. Yani beni merge et diyemiyoruz. Github da ise durum farkli.
* Üst branch e gectikten sonra merge islemi icin hangi alt kolu su an bulundugumuz kola merge etmek istersek onun adini yazariz.
* Mesela su an biz master ana branchte bulunuyoruz diyelim ve ahmet isimli alt kolu merge ettigimizi varsayalim:
* git merge ahmet dememiz yeterlidir.
* Not: Eger master brancten yada main brancten actigimiz alt branchler de kendi icerisinde alt kollara ayriliyorsa, öncelikle en alt kollardaki branclar üst kollarina merge edilir. Sonrasinda ise master ve main branchin alt kollari master ve main e merge edilir. Yani silsile olarak alttan üste dogru gelir merge islemi. Ve böylece Ana branchimiz güncellenmis olur. Ancak söyle bir durumda imkan dahilindedir. Ana branchin 2 ,3 veya daha fazla alt kademesinde bulunan bir branch direkt olarak ana branch a merge edilebilir.
* **Not:** Mesela master ana branchindan ahmet adinda bir kol ayrildiginda, bu ahmet branchinde master da olan tüm dosyalar mevcut olur. Ama ayrilma isleminden sonra master da degisiklik oldu ise o zaman merge yapmadan ahmet e gecmez bu degisiklikler.

**Merge Conflict**

* Conflict sadece ayni dosyanin ayni satirlarinda birden fazla kisi tarafindan degisiklik yapilirsa o zaman olusur. Eger bir dosya üzerinde birden fazla calisan kisi varsa ve bu kisiler ayni satirda farkli degisiklikler yapmak isterse, merge esnasinda bu durum bir conflict olusturur. ve bu durum teamleader yada project manager tarafindan cözülür. Git bu durumu kendi kendine cözmez. Cözüm asamasinda ise teamleader hangi kodu kullanmak isterse onu alir ve digerini siler. Gerekirse iki kodu da alabilir.
* **Conflict giderme asamalari:**
* Merge komutu girildikten sonra conflict mesaji alinirsa, öncelikle herhangi bir editör kullanilarak dosya icerisine girilir.
* Daha sonra conflict meydana getiren satirlar düzenlenir ihtiyac olanlar alinir geriye kalanlar silinir ve dosya kaydedilip cikilir.
* Sonrasinda ana branchte iken tekrar git add ve git commit yapilir ve böylece merge islemi gerceklestirilmis olur.
* Not: Ayni dosyada calisan iki kisiden biri ilk satirda digeri ikinci satirda degisiklik yaparsa bu merge conflict olusturmaz.