GIT & GITHUB

- GIT bir versiyon kontrol sistemidir. örneğin oyunlarda kayıt yapıp ölünce geri oraya dönmek bir vers.kontr.sistemidir yani kısaca checkpoint yapmak. bu avantajlar kodlama yaparken özellikle grup çalışmalarında çok işe yarayacak.
- açık kaynak kodludur ve alternatifleri vardır ancak %90 bu sistem heryerde kullanılır
- komut satırı , shell , kabuk aynı anlamlara gelir

KOMUTLAR (İleri seviye komutların altlarda kendi alanlarında açıklamaları var !)

- `ls` → içinde olduğumuz klasörde ki klasör ve dökümanları gösterir , masaüstünde gezinmek gibi
- `pwd` → nerede olduğumuzu gösterir
- `cd (klasöradı)` → klasör yada dökümanlara gideriz
- `cd ..` → olduğumuz yerden bi kere geri gelir
- `mkdir (klasöradı)` → yeni klasör oluşturur
- `rm -rf (klasöradı)` → klasör siler
- `touch (dosyaadı)` → notdefteri filan dosya türünü oluşturmak için
- `rm (dosyaadı)` → dosyayı siler
- `git branch` → branchlerimizi gösterir
- `git branch “isim”` → branch oluşturma
- `git switch “isim”` → yazılan branche gider
- `git merge “isim”` → yazılan branche ana branch ile birleştirir
- `git stash` → çalışmakta olduğumuz dosyayı depoda saklar
- `git stash pop` → son saklanına geri getirir

- `git stash list` → saklama listemisi gösterir
- `git stash apply “stash adı"` → istediğimiz saklananı geri getirme (stash adı list de yazar)
- `git reset “hash”` → commiti siler dosyada ki içeriğini tutar (hashi girilen hariç)
- `git reset --hard “hash”` → commiti komple herşeyiyle siler
- `git revert “hash”` → hashi girilenin commiti tutar verilerini siler
- `git diff` → dosyada yapılmış değişiklikleri gösterir hash ve branchler girerek karşılaştırmada yapılır
- `git rebase master` → master commitlerini önce bizim bracnh commitleri sonra dizer
- `git remote add origin <url>` → gitgub deposu ile bağlantı kurar , urlye origin ismini verir
- `git remote` → bağlantıları listeler
- `git branch -r` → bağlatılı olduğumuz branchleri gösterir (bu branchlere geçmek için switch yerine checkout kullanılır aslında commite değiştirmişiz gibi oluruz detachedhead olur saten)
- `git push -u origin main` → main branchi commitler origine bağlı depoya atar
- `git push` → aynı branchde devam ediyorsak tekrar tekrar üsttükini yazmaya gerek yok
- `git pull origin master` → fetch + merge direkt commitleri çekip merge eder mastere
- `git clone “url”` → githubdan komple repoyu çekmek

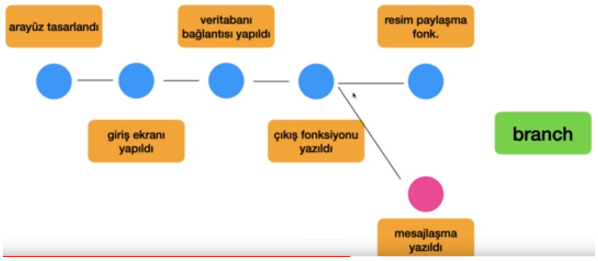
GIT KAYIT OLUŞTURMA

- gite email ve isim kaydetmek bu ilk başta sadece bilgisayarda tutulur ancak daha sonra ortak projelere bağlandığında kim hangi kodu yazdı ne zaman yazdı gibi bilgileri tutarken kullanılır
  - `git confih –global user.name “denbay”` → kullanıcı adını belirledik

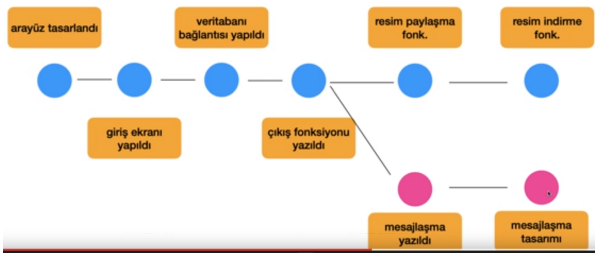
- `git confih –global user.email denizbyat@gmail.com` → kullanıcının epostasını belirledik

TEMEL YAPI

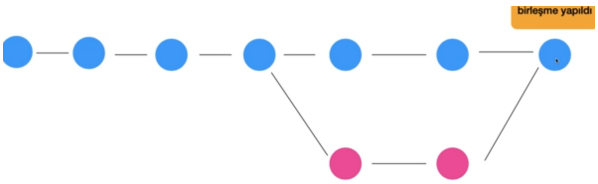
- Her bir işleme commit denir. bu işlem tamamlanıp projeye kayıt edilen her bir adımı gösterir



- branch dallanma anlamına gelir. burada proje bir koldan devam etmek yerine örneğin ekip halinde çalışırken diğer kişilerde istediği commit yerinden dallandırıp oradan başka bir kısmı yapabilir.

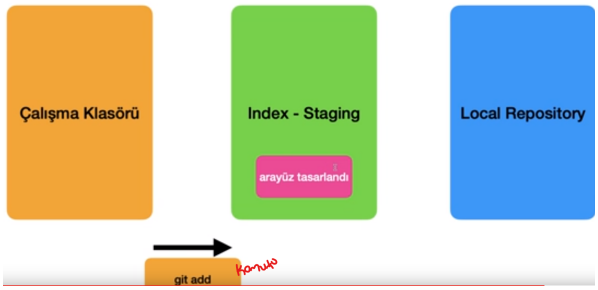


- branchler istenildiği kadar uzatılabilir

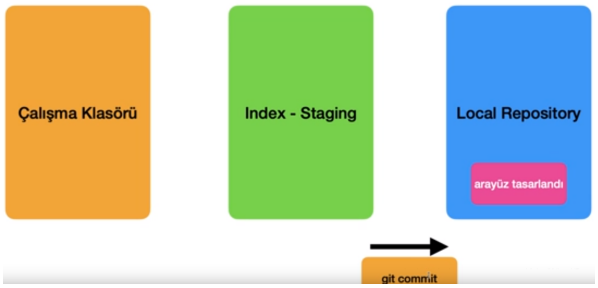


- ileride istenildiği yerde ana kola bağlanabilir

- commit oluştururken aceleci olmamak lazım. bunun için önce çalışma klasöründe çalışarak hazır hale getiririz



- hazır hale gelen ve ufak tefek pürüzleri kalan kısmı index yada staging denen yere `git add` komutu ile yollarız
- burada son kontroller yapılır



- son olarak hazır hale gelen komutu `git commit` diyerek projenin ağına ekleriz

COMMİT VE STATUS

- Kodlar filan yazarken bunu klasörler içinde yaparız. Bir klasör oluşturur ve içinde kodu yazacağımız döküman oluşturup yazarız. Bunu yaparkende klasörü gite bağlamamız gerekir bunu konsolda bi kaç işlemle yaparız.

- Klasörün içindeyken `git status` komutu ile klasörün git ile bağlantı durumunu kontrol ederiz. Gelen cevap yüksek ihtimalle herhangi bir git ile bağlantısı olmadığını söyliyecek. Eğer bi bağlantısı saten var ise ağının ismiyle birlikte ve commitleri gözükecektir bu durumda başka bir git ile bağlanmamalıdır.
- Bundan sonra git ile bağlantı sağlamak için `git init` (git initialized) yani giti başlat anlamına gelir. Başlatıldıktan sonra hemen ardından gitin özelliklerini ismini filan verecektir. Genellikle ismi “master(efendi)” olarak başlar bunun anlamı branchler gelince asıl ana ağın bu olduğunu gösterecek. Klasörün içinde .git adında gizli bir klasör oluşur ve burası repository olur tüm işlemler burada tutulacaktır. Gizli klasörler `ls` de gözükmez onun yerine `ls -la` yapılmalı
- Klasör içine touch ile dosyalar eklediğimiz de tekrar status yapınca farklı mesajlar gelecektir

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ilkdefter.txt
        ornek.py
```

- örneğin txt ve py dosyaları oluşturmuş olalım bunlar untracked (takip edilmiyor) halde başlar. Status yapınca bize gösterir saten. Yani bu dosyalar daha çalışma klasöründe bunları daha sonra git add ile indexe taşıyacağız. Taşımaz isek burada kalır ve takip edilemez olarak paylaştığımız zaman ziyaretçiler tarafından da gözükmez. Burası bize özel gizli alan gibidir.
- `git add “dosyaadı”` ile hepsini kaydeder indexe dosyayı attık bundan sonra bize oto olarak güncel statusu gösterecektir. `git commit -m “mesaj”` commit yaptığımız da tüm indextekileri respo ya atarcaktır. mesaj kısmı commit hakkında yani işlemi kısa birkaç kelime ile açıklama yazmak zorunludur ileride karışıklığı önler.
- `git add .` ve `git commit -a` tüm herşeyi bikerede sonrakine atar

LOG

- `git log` komutu ile commitler hakkında tüm bilgileri gösterir. Her commitin benzersiz hash'leri vardır bu tc gibi özel ifade commitle ilgili işlemler yaparken kullanırız.
- Kayıtlı dosyalarda herhangi bir değişiklik yaptığımız zaman otomatik tekrardan commit edilmeden önce ki yere indexe düşer ve yaptığımız işlemi yine mesaj ile commitlememiz lazım. 2.bir commitleme durumuda logda bu şekilde gözükecektir mesajlar altta gözükür. en üstteki güncel olan.

GİZLİ KLASÖR

- Ana klösör içinde .gitignore diye dosya oluştururuz ve içine görmezden gelinecek dosya isimlerini yazarız. örneğin Değişiklikler yaptıktan sonra tüm dosyaları commit etmek gerektiğinde -a yapınca orada bulunan dosyaları commitlemeyecek görmezden gelecektir. `touch .gitignore` dosya oluşturulur. dosyayı açıp içinede gizli.txt diye bi dosyamızında bize özek olmasını istiyoruz o zaman gitignore içine gizli.txt yazıcaz direkt. Bundan sonra statusde gizli.txt gözükmeyecek -ignore şeklinde tek gözükecektir.
- gizli.txt kodlar içinde filan yine kullanılabilecektir ancak github da filan paylaşım yapıldığında , onun içinde değişiklik yapıldığında log da gözükmeyecektir kısaca git takip etmiyecek bizim sorumluluğumuzda olacaktır.

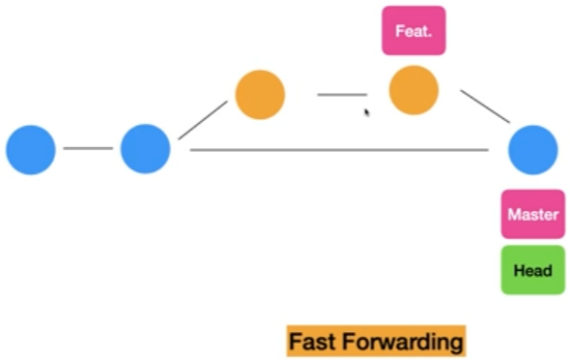
BRANCH

- Master default olarak ana branche verilen isimdir
- HEAD Güncel olarak en son nerede hangi commit de olduğumuzu gösterir , sürekli hareket eder bizi takip eder

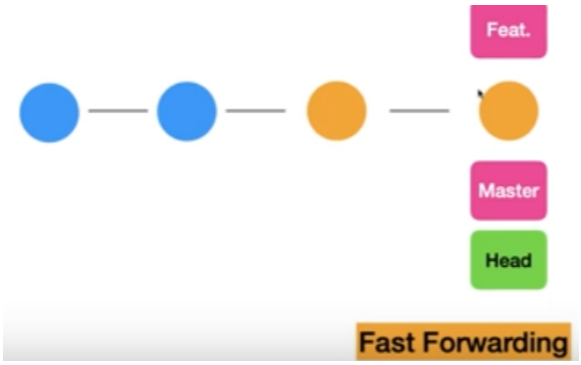
- Yeni Proje için : önce proje için klasör oluştur , status ile durumu kontrol et yeni olsa bile yinede bak , init ile git bağla , sonra dosyaları oluşturmaya başla , adım adım dosyalarda çalış , her adımdan sonra add ile indexe at sonra mesaj koyarak commitle <her adım da alışkanlık haline getir>

MERGE

- Branch birleştirme işlemidir.
  - `git merge "isim"` → yazılan branche ana branch ile birleştirir
- 1-Fast forwarding (ileri sarma) dediğimiz işlem genellikle deneysel bişey yaparken ve diğer branchlere zarar gelmesini önlemek için yapılan bişeydir. önce başka branchde deneriz eğer çalışıyor ise bağlarız.



- Birleşimden sonra commitleri ana branche koyacaktır çakışma olmadığından dolayı yani feat branchini yaparken ana branche başka bişey eklenmemiş ana branchdeymişiz gibi çalışmışız durumu böyle olunca da featdeki commitleri masterda ki sıraları boş olduğundan direkt mastera atar.



- 2-CONFLİCT (çakışma) fats forw. tersine birleştirme esnasında çıkan çakışma yani hatalardır. bu hatalar çok çeşitli karmaşık olabilir ancak bunu konsolda otomatik açıklamasını yapar aynı zamanda default olarak yazdığımız editörde de çakışmanın olduğu yerlerde işaretler , açıklamalar ile yardımcı olur

STASH

- Branchler arası yer değiştirirken bazı sıkıntılar çıkabiliyor Örneğin bir branch de çalışıyoruz ve daha commit edilecek kayda değer bişey yapmadık ancak başka bir branche geçip orada başka bişeyler yapmamız gerekiyor commit etmeden (çünkü öyle her zaman commit edilmez kayda değer bişey olmalı) switch yaparsak **1-** çalıştığımız dosyayı da eğer gittiğimiz branch de aynı isimli başka dosya yoksa bizimle beraber o branche getirir ve yanına “added” diye işaret koyar diğer dosyalardan farklı olduğu belli olması için ancak **2-** gittiğimiz branchde aynı isme sahip bir dosya var ise burada conflict oluşacaktır. işin kötü yanı geride kalan branchde ki dosyanın içeriğini alır yeni geldiğimiz branchde ki aynı isimli dosyanın içeriği ile değiştirir ve bu dosyanın içeriği silinir. Konsolda bize içeriği silinen dosyayının son commit edilmiş haline geri getilsin mi diye sorar `git restore “dosyaadı”` şeklinde. dosya son commit haline geri döner ancak bu seferde bu hali alır diğer branchde ki dosyaya da koyar ve diğer branch deki dosya bu sefer tamamen gitmiş olur commit yapmamıştık çünkü (add yapmak bişey ifade etmez commit olmuş olması lazım)
    - İşin 3 çözümü vardır 1-en başta dosyayı commit et , 2-restore kullanmadan önce veriyi kopyala , 3-stash kullan(KULLANILMALI)
    - tek yapmamız gereken switch yapmadan önce `git stash` komutu kullanarak çalışmakta olduğumuz dosyayı git kendi deposunda saklar. Diğer işleri hallettikten sonra geri kendi branchimizi dönünce sakladğımız dosyayı geri getirmek için de `git stash pop` komutu kullanırız ve son sakladığımız dosyayı geri getirir. `git stash list` ile saklananları görürüz ve `git stash apply “stash adı"` (listedeki ilk bilgeleri isimleridir) ile de illa son saklanılanı değilde istediğimizi geri getiririz

CHECKOUT

- Bir dosya üzerinde çalışırken dosyayı bozarsak `git status` yaptığımızda bize `git restore` seçeneğini sunacaktır yani bu bozduysan bir önceki commite dönerbilirsin anlamına gelir.
- Eğer oluşturduğumuz bir commite sonradan dönüp tekrar düzenleme yapmak istersek eğer `git checkout "cmthashi"` komutu kullanırız. commit hashini `git log` kısmından bulabiliriz. Bunu yaptığımız da bize bazı mesajlar verecektir :
    - Bunu yaptığımız da bizim nerede olduğumuzu gösteren HEAD döndüğümüz commite gelecektir ve masterden ayrılacaktır. Bu durum Detached Head olarak adlandırılır. yani head koptu dikkatli ol buralarda demek ister. Biz döndüğümüz de log çalıştırınca olduğumuz yerden sonraki logları göremeyiz (editörde gözükürde o editörün olayı gitle alakalı değil save etmeden kapatabilirsin). 2 seçenek var : (bunları konsol saten kısaca gösterir)
        - 1-Döndüğümüz committe notlar alıp , kopyalama filan yapıp mastere geri dönebiliriz dikkat değişiklik yapmadık. bu kolay yoldur direkt `git switch master` yaparız ve kaldığımız yerden devam edebiliriz ve head artık masterla aynıdır buna da Reattached Head denir.
        - 2-Döndüğümüz committe değişiklikler yapacaksak eğer o committen yeni bir branch oluşturmamız gerekir. Buda normal branch oluşturur gibi masterdan farklı bir yoldan devam ederiz.

RESET

- commit silme işlemidir. `git reset "hash"` ile hashi yazılan commite kadar (o commit hariç) tüm commitleri sondan gele gele siler ancak commitlerde yapılmış değişiklikleri dosyalardan silmez. Eğer commitlerin dosyalarda yapılmış değişikliklerini de silmek için `git reset –hard "hash"` kullanılır. (dikkat o commitler ile eklenmiş dosyalardaki verilerinide siler)

REVERT

- revert reset ve checkout işlemlerinin kombinasyon hali gibidir. `git revert "hash"` hashi girilen commitin verilerini siler commiti silmez. Örneğin bir commiti

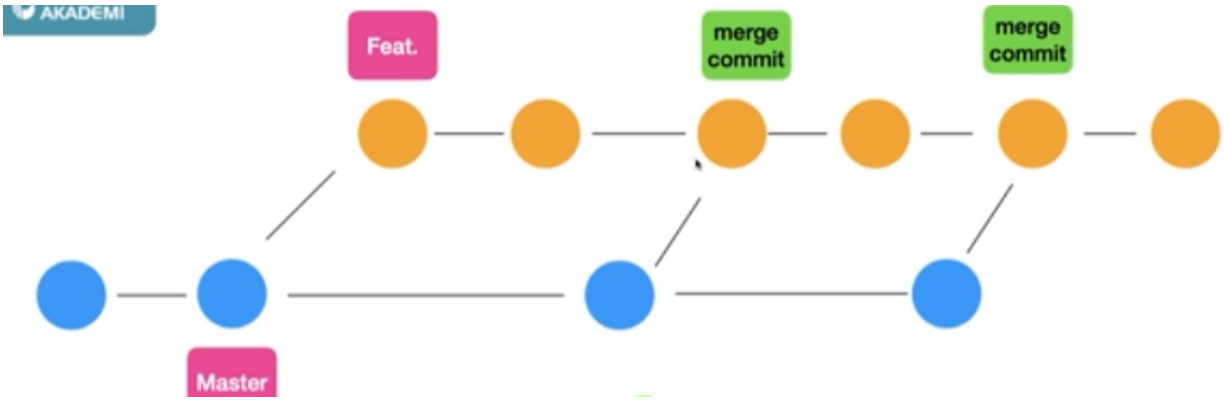
silmeden o commitin dosyada ki verilerini silmek isteyebiliriz yani commit dursun ancak onu kullanmıycam gibi. bu işlem takım çalışmalarında çok işe yarar. mesela projede branchler birleştirilmek istendiğinde ana commitler birbiri ile aynı olmalıdır işte bu yüzden bizde kullanmak istemediğimiz commiti silmek yerine onun verilerini bizden siliyoruz.

DİFF

- `git diff` dosyada yapılmış olan tüm değişiklikleri yada `git dif “hash” “hash”` iki commit arasında ki farkları yada `git dif “branchad” “branchad”` branchler arası farkı görmek için kullanılır

REBASE

- master (ana) branchden farklı bir branchde çalışırken ana branche commitler gelirse eğer o commitleri kendi branchimize merge edip çalışmamıza devam etmemiz gerekir ki ileride birleştirdiğimiz zaman sorun çıkmasın. Ama ana branche sürekli sürekli commitler gelebilir biz çalışırken ve biz bu kommitleri kendi branchimizi her aldığımız da bizde de yeni branchler oluşur ve bizim branch karışık hale gelmeye başlar



- bu karışık durumu çözmek için kendi branchimizdeyken `git rebase master` komutunu kullanırız ve bu komut branchimizin commit sırasını değiştirerek önce masterdaki commitler sonrasında da bizim commitler olarak sıralar ve böylece sürekli merge commit oluşturmak zorunda kalmayız.

- DİKKAT Bu işlemi yaparken sizin branchiniz daha önce başka bir yerde paylaşılmamış yani başka birileri tarafından kullanılmamış olması gerekir. Çünkü commitler normalde tarih sırasına göre sıralanır doğal olarak önce yazılandan sonra yazılana doğru ancak rebase yaptığımızda master commitlerinin önceliğine göre sıralar ve tarihleri şaşmış olur. branchimizi kullanan biri ile tekrardan paylaşımda bulunduğumuz zaman onun projesini dağıtır. Bu işlem nadiren kullanılır

GİTHUB

- github bir nevi sosyal medya gibidir ve git ile alakası yoktur diyemeyiz ancak çok farklı şeylerdir. Yaptığımız projeleri,çalışmaları burada özel yada herkese açık şekilde depo edebiliriz. En önemli özelliği ise git ile çalışırken yazdığımız commitler filan hepsini `git push` komutu ile bu depoya doğrudan aktarabiliriz.
- Depo oluşturduktan sonra bize depoya ait bir url tanımlanır.
  - Önce bu url yi git ile bağlamamız gerekir onun için `git remote add origin <url>` komutu çalıştırırız ve git hesabımıza giriş yapmamızı ister yaptıktan sonra bağlantı oluşur. `git remote` ile bağlantıları görebiliriz. [*remote* → uzak bağlantı anlamına gelir , *add* → ekle , *origin* → bizim urlyi her seferinde girmemek için urlye verdiğimiz isimdir bu isteğe bağlıdır genelde origin kullanılır]
  - Bağlantı oluştuktan sonra `git push -u origin main` komutu ile de varsa olan commitleri filan herşeyi depoya aktırır ve sonrasında yazılacak olan commitleri de aktarmak için bu komut kullanılır. [*push* → ittirmek-koymak anlamına gelir , *-u* → upstream demektir yani kaynak yönünü gösterir anlamına gelir bunu ilkinde birkez kullanınca işlemi kayıt eder ve artık sadece git push kullanmamız yeterli olur , *origin* → urlye verdiğimiz isim , *main* → bizim bulunduğumuz branchin adıdır genelde git de master olur default olarak github da maindir fark etmiyo git de branch adımız neyse onu yaz]. Bu komut kısaca bir kez yazıldıktan sonra artık `git push` olarak kullanılabilir ancak aynı branchde devam ediyorsak.

    - Push yaptıktan sonra githuba commitler eklenince github logunda o commitleri atan kişi olarak gitde ki kullanıcıyı baz alır yani github hesabını değil gitden kimin attığını gösterir.

- *Compare & Pull Request* : genellikle ana branchden başka branchler açtığımızda bildirim olarak githubda çıkar. Bunun anlamı kodların branchler arasında uyumlu olduğunu ve merge yani birleştirilebileceğini söyler saten birleştirirken de bizden yorum ister karışıklık olmasın diye. Birleştirme yapıldıktan sonra üst menüde Pull Request olarak bildirim düşer ve onay ister onuda onaylayınca yani merge edince artık birleştirdiğimiz branchle işimiz kalmaz ve sil diye seçenek çıkar silince de o branch artık gider.
    - DİKKAT şimdi bunları githubda yaptık ama gitde bunlar birleşmiş olmaz yani normalde gitle önde gidip githuba pushlardık şimdi bunu yapınca github gitin önüne geçti ve senkron sorunu oluştu. Bu sefer githubdan gite verileri getirmek için yani pushun tam tersi işlem için `git fetch origin master` kodunu kullanırız. böylece originin bağlı olduğu depodan masteri bizim masterle eşitler. [fetch → getirmek anlamına gelir] `git pull origin master` ise direkt değişiklikleri alır ve mastere merge eder pull = fetch + merge anlamına gelir. fetch daha güvenlidir önce getirirsein bi bakarsın sonra merge edersin ama pull direk merge eder

- *Clone* : githubda beğendiğimiz projeleri bilgisayara almak için önce o reponun urlsini almalıyız bu normal tarayıcıda ki url sonuna .git ekle yada sol üst köşede code tuşuna tıklayınca urlyi hazır şekilde gösterir. onu kopyalıyoruz ve konsola `git clone “url”` çalıştırdığımız da projede ki tüm dosyaları o anda konsolda bulunduğumuz yere indirir

- *Fork* : çatallandırma anlamına gelir ve projenin sağ en üsttünde bulunan butondur. bunun için koda filan gerek yok direkt basınca depoyu bizim hesabımızın deposuna alır ve artık ben bunun üstünde tek çalışıcam anlamına gelir yani o dosya ile yolları ayırmış oluruz artık proje bize aitmiş gibi olur. ancak buraya istediğimiz gibi eklemeler filan yapınca üstte bildiri çıkar ve forkladığımız orjinal halinden ne kadar önde olduğumuzu gösterir ve pull request yapmak istermisin yani projeye istek göndermek istermisin bu geliştirmelerini diyede sorar onlar yeni şeyler yaparsa eğer

yanında butonda gözükür hep kendimizi fetch ve merge yapabiliriz güncellemek gibi.

    - forklanan proje sahibi tarafından görünür.

- *Private Repostroy* : gizli depolarımızdır. belli kişile erişim izni vermek için deponun ayarlardan collaborators yani işbirlikçilere o kişiler eklenmelidir eklenince onlara istek gider. bu yetkilendirme işlemi tamamlanınca depoyu artık işbirlikçileri de gitden sahibi gibi değişiklik yapma izni verilir. işbirlikçiler değişiklik yaparken genelde kendine branchler oluşturup yine pull request sistemi ile eklemeler yapılır herkes herkesin kodunu önce test etsin sonra eklemek daha mantıklı.

- ReadMe.md : bu kısıma gerek yoktu ancak güzel büyük projeler yaparken kullanılır ve html gibi çalışır. Kendine ait kolay bi dili vardır readme yazarken readme.md style diye google de aratınca orada tüm özellikleri çıkar.

- IDE : kullandığımız editörlerden de git sistemine konsol gibi doğrudan erişim sağlayabiliriz. önce ide yi menüsünden git bağlarız ve bağlantıdan sonra konsolda komutlarla yaptığımız herşeyi orada menü halinde görürüz ve yaptığımız herşey aynı anda konsolda da yapılıyouş gibi olur bi ordan bi ordan yapıladabilir yani.