Mac’de iki adet Shell tipi varmış. Shell dediği command prompt’a tekabül ediyor.

1- Bash

2- ZShell

Windows’ta 3 tane var:

1- Command Prompt (Windows Shell)

2- Powershell

3- GitBash

Commands:

pwd -> print working directory

ls -> list item

ls -s -> list with sizes.

cd -> change directory

cd .. -> 1 üst folders git

clear -> terminali temizle.

mkdir -> make directory

touch -> create a file. touch index.html

rm -> removes a file (not folder!). Trash’e bile gitmiyor komple siliyor.

rmdir -> removes a directory (içi dolu olduğunda uyarı veriyor ve çalışmıyor)

rm -r [directory] -> removes a directory with content.

man ls -> ls commandinin manuel’ini açıyor.

cp -> copy. cp data/core.py queries/ (slash’lere dikkat etmek gerekiyor)

cp -r -> r stands for recursive. copies all the content.

mv -> moves. mv data/core.py queries/

Git Commands:

git –version

git status : o folderdaki son durumu ve hangi branchte olduğunu söylüyor.

git config –list: config’i gösteriyor.

git init: Initializing the folder.

git add: Adding the file to the staging area.

git add . :Adding all the files to the staging area.

git commit -m “some message”

git config –global user.email “some email address”

git config –global user.name “some name”

git log: o branch’teki commit history’i gösteriyor.

git checkout [commit\_id]: geçmişteki bir commit’e (ilgili commit\_id’ye tabi) gidiyor. işin garibi burdan sonra git checkout master deyince tekrar geri geliyor.

git checkout branch\_name: quotation mark vermeden yazınca o branche geçiyor.

git branch: branch listesini ve o an hangi branchte olduğunu gösteriyor.

git branch branch\_name: Yeni branch açıyor. quotation mark vermeden yazmak gerekiyor.

git checkout -b branch\_name: yeni branch açıp o branche geçiyor.

git merge second\_branch: master’dayız, second\_branch’teki ekstra değişiklikleri master’a aktarmak istiyoruz. Bu komutu kullanıyoruz.

git switch branch\_name: o branche geçiyor.

git switch -c new\_branch\_name: o branchi yaratıp oraya geçiyor.

git ls-files: files in the staging area.

git rm file\_name: diyelim bi dosyayı sildik, bu silinen dosya staging areadan kaybolmuyor, bunu oradan da silmek için git rm yapmamız lazım.

burası unstaged change’leri undo etmek için:

git restore file\_name: burada olay şu, diyelim bi dosyayı modify ettik, ama bunu staging areadan çıkarmak istiyoruz (neden basitçe silmiyoruz onu anlamadım). bu komutu çalıştırınca en son committeki haline dönüyor repo.

git clean -dn: diyelim repoya bi dosya ekledik, ama bunu silmek istiyoruz, -d delete anlamına geliyor -n listele anlamına geliyor. bu komut sadece ben bunları silerim anlamına geliyor, silmiyor. Silmek için :

git clean -df: yukardaki komutun sileni.

burası staged change’leri undo etmek için: diyelim bi dosyayı git add ile staging areaya ekledik sonra vazgeçtik.

git checkout file\_name → git reset file\_name (2 komut gerekiyor)

ya da: git restore –staged file\_name

commit silmek:

git reset –soft HEAD~1 : diyelim bi dosya yarattık bunu git add git commit diyerek ekledik. sonrasında da bu dosyayı silmek istedik. bu komut 1 commit geri gitmeye yarıyor. resetten sonra dash dash. Fakat bu komut dosyayı silmiyor, dosya staging area’da kalıyor.

git reset HEAD~1: bu durumda dosya staging area’dan da silinmiş oluyor. (kontrol etmek için git ls-files)

git reset –hard HEAD~1: bu arkadaş dosyayı da siliyor komple.

branch silmek: git branch -d (-D forced version) branch\_name

HEAD konsepti:

diyelim masterda 3 tane commit var ve biz masterdan başka bi branche checkout yapıcaz. Bu 3 commitin sonuncusu HEAD olarak geçiyor. Anladığım kadarıyla branchin son commitine HEAD deniyor.

Detached HEAD: sanırım bir branchin ortasındaki bi yerdeki commit’e gittiğimizde detached HEAD state oluyor.

Creating the repo.

Folder’ı repo’ya çevirdiğimizde .git klasörü açılıyor klasörde. Burada değişiklikler saklanıyor.

Folder’da iki alan var.

1- Staging: dosyalarda değişiklik yaptığımızda git add komutuyla dosyayı staging area’ya taşıyoruz. Bu şu demek, Git bak kardeşim ben bu dosyada değişiklik yaptım bunu gör. Heralde bu github desktop’taki checkbox’lara tekabül ediyor.

2- Commits: Burada da snapshot alıyoruz.

Branches: Bunun mantığı çalışan şeyi bozmamak. Yapacağın değişiklikleri önce paralelde bir kopya üzerinde yapıyosun sonra eğer çalıştığından eminsen main branch ile birleştiriyosun.

.gitignore: bu dosyanın commit edilmesi lazım öncelikle. içine şöyle şeyler yazabilirsin:

file\_name

\*.log #tüm .log ile biten dosyalar.

!sample.log #bu hariç tüm .log ile bitenleri ignore et.

folder\_name/\* #bu şekilde bu klasördeki tüm dosyaları ignore ediyosun.

buraya kadar olan kısmı 49. dersin sonundaki assignment’ın çözümünde tekrar edebilirsin.

**Dive-deeper:**

git stash: main branchte çalışırken yeni branch açmaya üşendin çalıştın çalıştın, bu git stash sayesinde yaptığın değişiklikler ram’de bi yere kaydediliyor ve HEAD’e geri dönülüyor,

sonra sen **git stash apply** diyince değişiklikler geri geliyor.

**git stash list** diyince de eğer birden fazla stash yapmışsan bunları listeliyosun.

**git stash push -m “some\_message”** diyerek açıklama girebiliyosun.

**git stash drop index\_number** diyerek de stashi silebiliyosun.

**git stash** **clear** diyerek hepsini silebiliyosun.

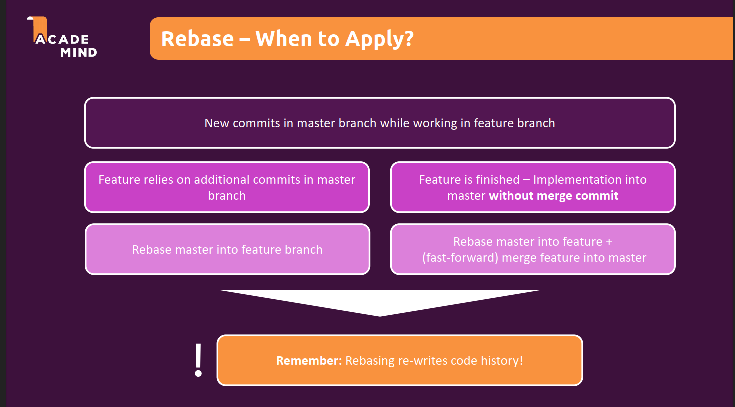
**git reflog**: diyelim bi commit yaptın sonra da git reset diyerek bu commiti ve yeni oluşturduğun her şeyi sildin. bu komut bunları geri getirmeye yarıyor. Galeri uygulamasındaki trash klasörü gibi 30 günlük rolling olarak tutuyor. şöyle çalışıyor: git reflog diyosun, burda arşiv gibi bişey çıkıyor, dönmek istediğin commit’in id’sini kopyalıyosun, git reset –hard [commit\_id] diyosun, oraya dönüyor. eğer branch sildiysen biraz daha farklı, yine git reset [commit\_id] yapıyosun ama bu sefer uyarı geliyor çünkü detached head state’tesin çünkü öyle bi branchin yok, burda da git switch -c [branch\_name] yapıp yeni branche bu içeriği kopyalıyosun.

**merging:**

**1- fast-forward merge:** diyelim master’dan bir branch açtın, çalıştın çalıştın yeni commitler yaptın. eğer ki master branch’te yeni commitler yoksa fast-forward merge yapıyosun.

**git merge –squash:** diyelim yeni branch açtın master’ı ellemedin, 2 tane dosya ekledin yeni branchte, eğer sonra git merge –squash [diğer\_branch] dersen yeni branchteki değişiklikleri tek bi parça olarak alıyor ve sadece staging areaya ekliyor, sonra sen tek bi commit yaparak yeni branchteki tüm commitleri tek bi committe birleştiriyosun.

**2- recursive merge (non-fast-forward):** yine bi master branchin var, bikaç commit sonra feature branchi açtın, feature branchinde bikaç commit yaptın, bu sefer diyelim master branchte de bikaç commit yapıldı, yani artık doğrudan merge yapamayız. bu durumda recursive kullanıyoruz. aslında yaptığımız ekstra bişey yok yine git merge [branch\_name] diyosun, sadece git recursive şekilde yapıyor.

**3- rebase:** diyelim master’dan yeni bi feature diye bi branch açtık, sonra masterda bi commit var, feature branchinde de commitler var, eğer rebase yaparsak featuredaki commitler farklı hashler ile masterdaki commitin üstüne ekleniyor, içerik tabi aynı kalıyor. komut: **git rebase master,** feature’ın üzerine master’ı ekledik. Burada önemli bir nokta var, commit id değiştiği için aslında burda historyi tekrar yazıyoruz gibi bir durum var, collaboration anlamında sıkıntı yaratabilir dikkatli kullanmak gerekiyor.   
  


**Handling the merge conflicts:**

iki branchin aynı dosyasında farklı şeyler var, diğerini öbürüne merge etmek istediğinde hata çıkıyor, bi şekilde eşlemen lazım, bu noktada VS Code’un güzel özellikleri var ikisini de gösterip bunda bu var diğerinde bu var hangini tutayım diye soruyor. komutlar:

git log –merge : burda commitleri gösteriyor

git diff: bu da iki commitin arasındaki farkları gösteriyor.

**Cherry picking:** diyelim bişeyleri commitledik sonra baktık typo var düzelttik bi daha commitledik. Bunu forklayacak adam gidip tüm commitleri almak istemiyor sadece belli başlılarını almak istiyor, bunun için bu özelliği kullanıyor. bunda da yine yeni bir commit\_id atama durumu var dolayısıyla dikkatli olmak lazım. komut: git cherry-pick [commit\_id]

**Tagging:** bazı commitlere tag atayabiliyoruz: git tag [tag\_name] [commit\_id] sonra da bunu git show [tag\_name] şeklinde görebiliyoruz. git tag -d [tag\_name] şeklinde de siliniyor. git tag -a [tag\_name] -m [commit\_message] şeklinde annotated da yapabiliyoruz.

**GitHub**

Local’den cloud’a geçiyoruz.

diyelim local’de bi folder’ımız var bunu github’a eklemek istiyoruz. Komut:

**git remote add origin URL.**

buradaki origin kelimesi “name of the remote machine” diye geçiyor. URL’in kısaltması aslında, uzun uzun url yazacağına origin yazıyosun.

**git pull / git push** remote repoya bişey göndermek / çekmek için kullanılıyor.

diyelim lokalden remote’a bişeyler göndericez, ilk önce repository’i github’da oluşturmak gerekiyor. adam lokaldeki folderın ismiyle repo’nun adını aynı yaptı, convention heralde. sonra o folderın içine terminalde gidip şunları yapıyor:

1- git init

2- git add .

3- git commit -m “commit message”

4- git remote add origin <https://github.com/gokhanakay/repo_name.git>

5- git push -u origin master (-u: upstream) bu sonradan açıklanacak.

5. maddeyi yapınca bi popup çıkıyor sign-in olmak için. burada personal access token’la da giriş yapabiliyosun.

Settings -> Developer Settings -> PAT -> Generate new token.

İkinci bir commit yapmak istediğimizde, commit’ten sonra sadece git push dersek hata alıyoruz (this branch is not an upstream branch gibi bişey), bunun yerine git push origin master dememiz gerekiyor.

**git branch -a :** remote dahil tüm branchleri görebilmek için

**git branch -r:** sadece remote repo’ları gösteriyor.

Diyelim remote’ta yeni bi branch açtık, bunu lokal doğrudan göremiyor.

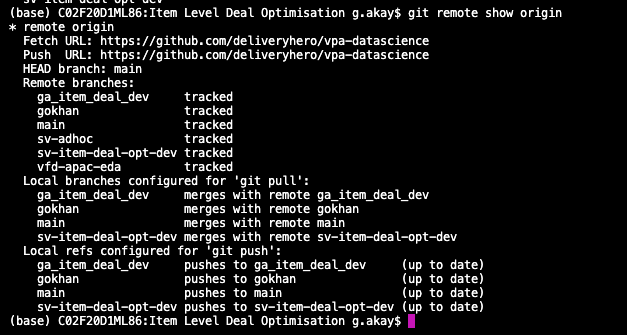
**git ls-remote** komutuyla remote’taki güncel branchleri görebiliyoruz.

**git fetch** **origin:** dersek artık lokalde yeni bir branch yaratıldığını görebiliyoruz **git branch -a** komutuyla.

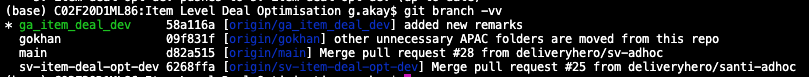
**git pull = git fetch + git merge**

sonra git pull ve git push’un arkasında dönen şeyleri anlattı, tracking branch olayları var, gerçek hayatta kullanılmıyor ama bilseniz iyi olur diyor. Şu komut çok önemli:

**git remote show origin:** bu arkadaş lokal ve remote arasındaki köprüleri gösteriyor.

****

**git branch -vv:** bu da lokaldeki branchler remote’ta hangi branchleri takip ediyor onu gösteriyor.



**git clone:** öncesinde bi tane klasör yaratıyoruz. sonra terminale gidip git clone URL diyoruz. Done. Sonra diyelim bu klasörde yeni dosyalar yarattık. öncesinde branch açtık tabi. bunu remote’a pushlamak için:

**git push origin [yeni\_branch\_name]**

**İŞİN ÖZETİ:** diyelim yeni bi dosya ekledin commitledin remote’a atmak istiyosun:

**git push -u origin [new\_branch\_name]:** bu şekilde yaptığında tracking işini de ayarlıyor.

lazım olursa diye: **git branch –track branchname origin/branchname**

**GITHUB COLLABORATION**

git clone yapmak için ilk önce klasörünü yaratıp sonra o klasöre terminalde ulaşıp orada **git clone** **URL** komutunu çalıştırıyoruz.

bir repoyu clone ettik diyelim, değişiklik yaptık commitledik, sonrasında push’lamak için PAT istiyor. repo sahibinin PAT vermesi gerekiyor user’a.

**Forking**: Git clone yaptığımızda repo’yu local’e çekiyoduk, sonrasında değişiklikler yaptığımızda bunu commit ediyorduk, fakat bunun için collaborator olmaya ihtiyacımız vardı. Fork yaptığımızda ise kişilerin repo’larını kendi remote’umuza kopyalıyoruz, lokale değil. Hatta bunun git ile de alakası yok, bu github’ın bir feature’ı. Diyelim forkladığımız bir repoda geliştirmeler yaptık, eğer bunu asıl repoya iletmek istiyorsak **Pull Request** açıyoruz, ve asıl repo sahibine diyoruz ki bak kardeşim böyle böyle değişiklikler yaptım istersen bunları al. Kabul edip merge’lemesi lazım bu değişiklikleri.