**GIT (VERSİYON KONTROL SİSTEMİ)**

**TEMEL BİLGİLER**

**Git**, Linus Torvalds (Linux'u yaratan kişi) tarafından geliştirilmiş bir versiyon kontrol sistemidir. Versiyon kontrol sistemi nedir? Geliştiriciler yeni bir proje yaptıklarında, kodlarına daima güncelleme yaparak devam ederler. Versiyon kontrol sistemi, kod temeline yapılan değişikliklerin izini sürmeye yardımcı olur.

Basitçe anlatmak gerekirse: bir dosyanız var ve bu dosyanın oluşturulma tarihinden başlayarak istediğiniz tüm durumlarını takip edebildiğiniz bir yapı tasarlıyorsunuz. Bu yapının sorun çıkarıp çıkarmaması tamamen bizim elimizde. Ne kadar iyi tasarlarsak ne kadar çok commit atıp geri dönebileceğimiz noktalar oluşturursak, yönetimi kolay bir yapı tasarlamış oluruz. Kısaca yazılım projemizin tüm anlarını kayıt altına alıyormuşuz gibi düşünebiliriz.

**Repository kavramı:**

Versiyon kontrolünün en temel bileşeni repository denilen yapıdır. Repository, dosyalarınızdaki tüm değişiklikleri ve bu değişiklikler ile ilgili ilave bilgileri (değişikliği kim, ne zaman yaptı ve değişiklik ile ilgili girilen açıklamalar) ayrı birer versiyon olarak kayıt altında tutan bir veri tabanıdır. Git tüm bu bilgileri genellikle dosya sisteminde gizli bir klasör olarak oluşturulan .git isimli klasör içinde bir dizi dosya olarak tutar.

Bu bahsettiğimiz repository'yi kendi bilgisayarınızda oluşturmak için iki yöntem kullanabilirsiniz.

* Henüz versiyon kontrolünde olmayan bir projeniz varsa **git init** komutu ile projenizi tüm klasör ve dosyaları ile birlikte versiyon kontrolüne alabilirsiniz.
* Projeniz uzaktaki veya şirket ağınızdaki bir Git sunucusunda versiyon kontrolü altında tutuluyorsa projeyi kendi bilgisayarınıza **git clone** komutu ile indirebilirsiniz.

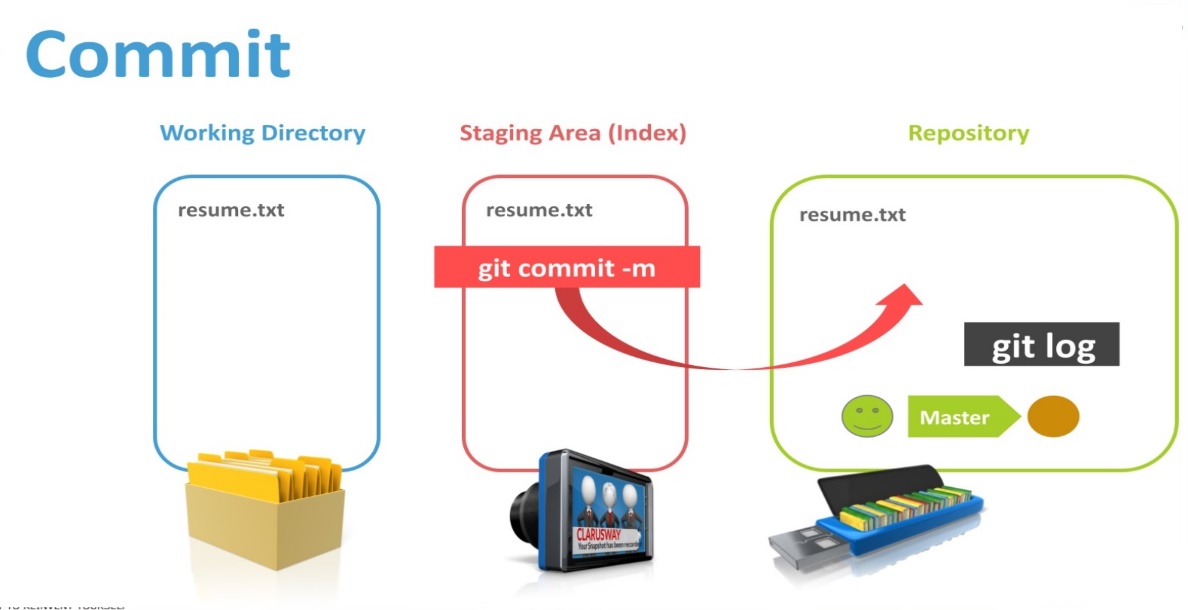
Git reponuzda dosyanız üç ana durumda bulunabilir: Modified, Staged ve Committed.

**Commit:**

Git'e eklediğimiz dosyaları kalıcı olarak Git veri tabanına işlemeye commit denir. Proje için yaptığınız işlemleri her defasında commit etmek gerekiyor.

**Commit ne işe yarar:**

Diyelim ki projenizi geliştirirken bir yer patladı veya önceden yazdığınız bir kısım, şimdi eklemek istediğiniz bir özellikle ters düştü. Bu durumda dönüp onlarca dosyadan yüzlerce satırı silmekle uğraşmanız gerekecektir. İşte bu durumda aldığımız commitler bizim hayatımızı kurtaracaktır. Git sayesinde istediğimiz zaman istediğimiz commite geri dönebiliriz. Projemiz bugün patlasa, biz dün aldığımız commite veya geçen ay aldığımız commite veya projenin çalışan son haline geri dönebiliriz.

****

**Git status**

Commit işlemi öncesinde dosyalarınızda yaptığınız değişikliklerin bir özetini görmek isteyebilirsiniz. git status komutu ile hangi dosyaları değiştirdiğinizi, sildiğinizi veya hangi dosyaları eklediğinizi kolayca görebilirsiniz.

Bir sonraki aşamada değişen dosyalarınızdan hangilerinin commit'e dahil olduğunu belirlemeniz gerekiyor. Bu adımda commit'e dahil etmek istediğiniz dosyaları staging area denilen ara bir alana alırız.

Dosyalarınızı staging area'ya ekledikten sonra şimdi commit işlemine hazırsınız. Commit işlemi ile dosyalarınızdaki değişiklikler yeni bir versiyon olarak Git'de kayıt altına alınır.

Zaman zaman, özellikle de bir takım çalışması söz konusu ise, projenizdeki değişikliklere göz atmak isteyebilirsiniz. Projeniz için oluşturduğunuz commit'lerin tarihçesini incelemek için *git log* komutunu kullanabilirsiniz.

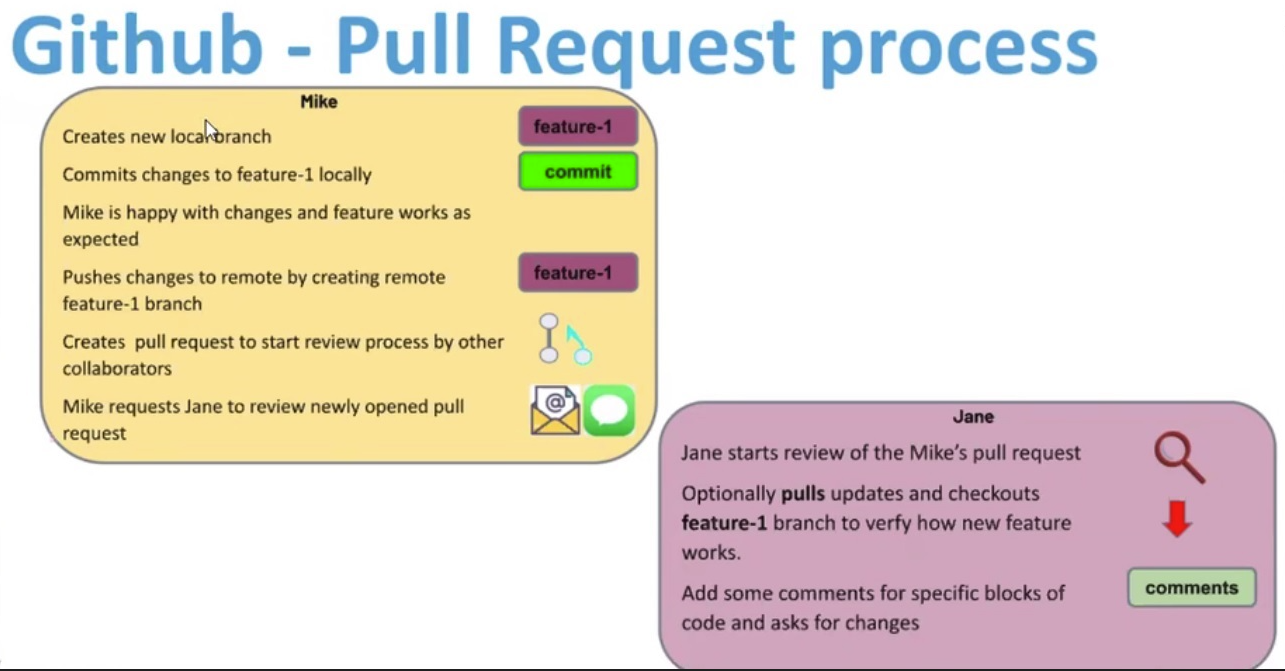
Yaptığınız değişikliklerin takımın geri kalanı tarafından da görülmesini ve kullanılmaya başlanmasını sağlamak için değişikliklerinizi zaman zaman uzaktaki repository’de yayınlamanız gerekir. Bunun için *git push* komutunu kullanırız.

**Branch**

Yeni bir fikrimiz olduğunu varsayalım, ancak bundan tam olarak emin değiliz. Bir şey denemek istiyoruz. Bu durumda yeni bir branch oluşturabilir, biraz ilerleme kaydedebilir ve yeni branch’tan memnun kalmazsanız ana (main) branch’a geri dönebilirsiniz.

**Pull request**

Projeye diğer katkıda bulunanlara veya ekip üyelerine, repo üzerinde bazı değişiklikler (hataları düzeltme, yeni özellikler ekleme gibi) yaptığınızı ve bunları repoya katmak istediğinizi söylediğiniz bir yöntemdir. Bu değişiklikler yetkili kişiler tarafından kontrol edilip onaylandıktan sonra pull yapılıp (çekilip) repo ile birleştiriliyor.



**Local (Yerel) & Remote (Uzak) Repository'ler**

* **Local repository**: Kendi bilgisayarınızda proje klasörünüzün altında bulunan .git klasörüdür. Bu repository üzerinde sadece siz çalışabilirsiniz ve değişiklikler yerel diskinize kaydedilir.
* **Remote repository**: Genellikle uzaktaki bir sunucuda yer alırlar ve bu sunucudaki .git klasöründen ibarettirler. Takım çalışması söz konusu ise takımdaki kişiler değişikliklerini bu uzaktaki repository üzerinden paylaşırlar.

**Local bir proje oluşturmak:**

**Henüz version kontrolü altında olmayan bir projenizi versiyon kontrolü altına almak** **için** **git init** komutunu kullanırız. Bu işlemi gerçekleştirmek için Mac OS X'de Terminal uygulamasını Windows'da ise Git Bash'i açarak aşağıdaki komutları çalıştırmanız gerekir:

**$ cd proje/klasörünüzün/yolu/**

**$ git init**

Bu işlemden sonra:

**ls -la**

komutu ile proje klasörünüz altındaki dosyaları listelediğinizde klasörün içinde .git isimli gizli bir klasörün olduğunu göreceksiniz.

git init komutu ile projemiz için **boş** bir repository oluşturduk. Ancak proje klasörümüzde dosyalar ve başka klasörler bulunmasına rağmen bu dosya ve klasörlerin hiç biri henüz Git tarafından versiyon kontrolü altına alınmadı.

**İlk commitimiz:**

$ git add -A

$ git commit -m "İlk commit işlemimizi yaptık"

İlk komut ile tüm proje dosyalarımızı staging area'ya ekliyoruz.

İkinci komut ile dosyalarımızı bir açıklama ekleyerek commit ediyoruz.

Bu iki komutu arka arkaya kullanmak yerine aynı işlemi git commit -a komutu ile de yapabiliriz.

**Remote Repository’ler:**

Günlük çalışmamız sırasında staging ve commit gibi versiyon kontrolü ile ilgili işlemlerin çoğunu yerel diskimizde yer alan local repository üzerinde yaparız. Proje'de çalışan tek kişi siz iseniz muhtemelen Internet'de veya yerel ağıda yer alan remote bi repository oluşturmanıza da gerek olmayacaktır.

Ancak takım çalışması söz konusu olduğunda, takımdaki geliştiricilerin birlikte çalışabilmesi için herkesin değişikliklerini ortak bir alanda yayınlaması ve diğerlerinin de bu ortak alan üzerinden bu değişiklikleri kendi branch'lerine entegre etmesi gerekecektir. Bu durumda başvuracağınız en etkin araç Git'deki Remote Repository işlevleridir. Remote repository'leri en basit anlamda tüm ekibin erişimi olan dosya sunucusu olarak düşünebilirsiniz

**Local repo ile remote repo arasındaki fark:**

Teknik olarak remote repository'ler ile local repositoryler arasında bir fark yoktur. Local repository'ler için önceki bölümlerde ele aldığımız commit işlemi, branch oluşturma gibi işlemlerin tamamı remote repository'ler için de yapılabiliyor.

Ancak tüm bu benzerliklere rağmen remote repository'ler için Working Copy (aktif branch'deki dosyaların diskimizdeki kopyaları) yapısı geçerli değildir, remote repository'lerde sadece Git'in veri tabanının tutulduğu **.git** **klasörü** yer alır

**Repository oluştururkenki farklar:** Local bir repository ancak iki şekilde oluşturulabilir

* Boş bir repository olarak sıfırdan **git init** komutu ile oluşturabilirsiniz veya
* Remote bir repository'yi **git clone** komutu ile yerel diskinizde indirebilirsiniz.

Remote repository'ler de iki yöntem ile oluşturulabilir

* Local repository'nizi **git clone** komutunu **--bare** parametresi ile kullanarak remote bir repoository'ye klonlayabilirsiniz veya
* Boş bir remote repository oluşturmak için **git init** komutunu yine **--bare** parametresi ile kullanabilirsiniz.

**Remote bir proje oluşturmak**

Versiyon kontrolü Git ile yapılan bir projede yer alıyorsanız remote repository'lerinizi nasıl yöneteceğinizi de öğrenmeniz gerekir. Remote repository'leri projelerinizi internette veya sınırlı erişime izin verilen şirket ağında yer alan versiyonları olarak düşünebilirsiniz.

Diğer ekip üyeleri ile birlikte verimli çalışabilmek, onların yaptığı değişiklikleri kendi yerel çalışma alanınıza almak, kendi yaptığınız değişiklikleri onlar ile paylaşabilmek için remote repository'lerinizi doğru ve etkin bir şekilde yönetmelisiniz.

Git ile versiyon kontrolü yapılan bir projeye dahil olduğunuzda size verilecek ilk bilgiler :

* projenin Git adresi (URL) ve
* projeye erişim için kullanacağınız kullanıcı adı ve şifrenizdir.

Uzaktaki bir repository'nin (URL) adresi aşağıdaki formatlardan birinde olacaktır:

ssh://user@server/git-repo.git

kullanıcıadı@sunucuadı:git-repo.git

<http://example.com/git-repo.git>

<https://example.com/git-repo.git>

git://example.com/git-repo.git

Bu adres formatlarından ilk iki tanesi [SSH](http://en.wikipedia.org/wiki/Secure_Shell) (Secure Shell) protokolüne karşılık gelir. http:// ve https:// protokolleri ise normal internet erişimi için de kullanılan protokollerdir. Son format ise git'in kendi protokolüne karşılık gelir.

* Remote repository'nizin adresini,
* Kullanıcı adınızı ve
* Şifrenizi

öğrendikten sonra yapmanız gereken tek şey bu adresten projenizin dosyalarını yerel diskinize klonlamak. Bunun için öncelikle yerel diskinizde projenizi indireceğiniz bir klasör oluşturmanız ve Terminal'den bu klasöre gitmeniz gerekiyor. Sırasıyla aşağıdaki komutları Terminal'de yazınız

**$ cd Projects (**Projects klasörüne giriyoruz.)

**$ mkdir git101\_kitap** (git101\_kitap isminde klasör oluşturuyoruz)

**$ cd git101\_kitap** (git101\_kitap klasörüne giriyoruz)

Yerel diskimizde boş proje klasörümüzü oluşturduğumuza göre şimdi remote repository'mizi yerel klasörümüze git clone komutu ile indirebiliriz.

git clone komutundan sonra bir boşluk bırakıp klonlamak istediğimiz repository’nin linkini buraya yapıştırıyoruz.

**$ git clone** <https://github.com/clarusway/git101_book.git>

**Not:** Kullanıcı adınızı ve şifrenizi vererek remote repository'yi klonlamak için aşağıdaki **git clone** komutuna bu bilgileri aşağıdaki formatta vermeniz gerekiyor:

**git clone**[**https://kullanıcıadı:şifre@github.com/username/repository.git**](https://kullan%C4%B1c%C4%B1ad%C4%B1:%C5%9Fifre@github.com/username/repository.git)

**Projemiz üzerinde çalışmaya başlayalım:**

Üzerinde çalışacağımız projenin dosyaları artık yerel diskimizde yer aldığına göre projemiz ile ilgili normal çalışmamıza başlayabiliriz.

Projenizi ister local bir proje olarak oluşturmuş olun isterseniz remote bir repository'yi klonlamış olun tüm değişiklikleriniz yerel diskinizde gerçekleşecek ve **commit’leriniz** ile oluşturacağınız tüm versiyonlar git tarafından yerel diskinizdeki .git klasöründe takip edilecektir.

**git push** komutunu çalıştırmadığınız sürece yaptığınız değişiklikler sadece yerel diskinizde kayıt altına alınır.

**Dosya durumları:**

Git'de dosyalarınız genel olarak iki durumda olabilir

* Untracked (Takip Edilmeyen): Bu dosyalar versiyon kontrolü altında olmayan veya sizin henüz versiyon kontrolü yapmak için git'e eklemediğiniz dosyalardır. Bu dosyalardaki değişiklikler siz dosyaları git'e eklemediğiniz sürece versiyon kontrolüne tabi değildir
* Tracked (Takip Edilen): Bu dosyalar ise git'in versiyon kontrolü takibi altında olan dosyalardır. Bu dosyalar üzerinde yapacağınız tüm değişiklikler git tarafından takip edilmektedir

**Staging Area:**

Çoğu versiyon kontrol sisteminde değişiklikleriniz iki yerde kaydedilir

* Yerel diskinizdeki çalışma klasörünüz (working folder) veya
* Versiyon kontrol sisteminin veri tabanı

Ancak git'de değişikliklerinizin kayıt altına alındığı üçüncü bir alan daha vardır ki buna **Staging Area** denir ve git'in en temel kavramlarından birisidir.

Staging Area'yı, proje dosyalarımızdaki bir dizi değişikliği remote repository'ye göndermeden önce kayıt altında tuttuğunuz veri tabanı/alan olarak tanımlayabiliriz.

(Değişikliklerinizi commit etmeye karar verdiğinizde birbiri ile alakalı değişiklikleri tek bir commit olarak ele almaya özen gösterin. Birbiri ile alakalı olmayan değişiklikleri aynı commit ile versiyon kontrol sisteminde kayıt altına aldığınızda aşağıdakilere benzer sorunlar yaşama ihtimaliniz artacaktır

* Commit’inizdeki değişiklikleri inceleyen ekip arkadaşlarınız yaptığınız değişikliklerden hangisinin hangi konu ile ilgili olduğunu anlamakta güçlük çekeceklerdir.
* Alakalı alakasız değişiklikler tek bir commit içinde yer aldığı için herhangi bir nedenle belirli ve tek bir değişikliği geri almakta güçlük çekeceksiniz.

Alakalı alakasız değişiklikleri tek bir commit ile ele almak yerine örneğin iki ayrı sorunu gidermek için yaptığınız değişiklikler iki ayrı commit ile kayıt altına alınmalı veya daha büyük bir özellik üzerinde çalışırken bu özelliği oluşturan ve anlamsal bir bütün olarak ifade edilen daha küçük özellikleri de ayrı commit’ler ile kayıt altına almalısınız.

Projeniz üzerinde çalışırken belirli bir zaman aralığında yaptığınız değişikliklerin tamamının aynı konu veya özellikle ilgili olması mümkün olmayacaktır. Tam da bu noktada **Staging Area** mekanizmasının güzelliği ortaya çıkar, çünkü git hangi değişikliğinizin Staging Area'ya gideceğine karar vermeniz için sizin devreye girmenizi ister. Daha önce de belirttiğimiz gibi yaptığınız değişiklikler git tarafından otomatik takip edilmez, bunun yerine git tüm değişiklikleri sizin gözden geçirerek kontrollü bir şekilde Staging Area'ya almanızı ister.)

**Yaptığınız değişiklikleri listelemek:**

Son commit işleminizden sonra proje dosyalarınızda yaptığınız değişiklikleri listelemek için **git status** komutunu kullanabilirsiniz.

**$ git status**

**. git status**  komutu birlikte git, oldukça ayrıntılı durum bilgisi sunmaktadır

Bu komut ile git, aşağıdaki 3 ana grupta yer alan dosyaları size listeler

* Changes to be committed (Commit edilmeye hazır dosyalar): Bu gruptaki dosyalar **git add** veya **git rm** komutu ile Staging Area'ya eklediğimiz dosyalardır. Bu dosyalar bir sonraki commit'imizin içinde yer alacaktır
* Changes not staged for commit (Commit için henüz hazır olmayan dosyalar): Bu gruptaki dosyalar değişiklik yaptığımız fakat henüz Staging Area'ya eklemediğimiz dosyalardır. Bu dosyalar bir önceki grubun içine eklemediğimiz sürece bir sonraki commit'e dahil olmayacaklarıdır
* Untracked files (Versiyon takibinde olmayan dosyalar): Bu gruptaki dosyalar ise henüz versiyon kontrolü altına almadığımız dosyalardır.

**"git add" ve "git rm" komutlar:**

Bir önceki başlıkta değindiğimiz ve **git status** komutu sonrasında git'in bize özetlediği 3 gruptan son ikisinde yer alan dosyaların ilk gruba dahil edilmesi için **git add**ve**git rm** komutlarını kullanabiliriz.

Aşağıda oluşturduğumuz **git add** komutu ile **baslik\_2.md** ve **baslik\_2\_1.md** dosyaları ile **resimler** klasörü altındaki tüm dosyaların staging area'ya eklenmesini sağlayabiliriz. (Bunları teker teker de ekleyebilirsiniz.)

$ **git add** baslik\_2.md baslik\_2\_1.md resimler/\*

Benzer şekilde aşağıdaki **git rm** komutu ile **ornek2.md** dosyasının bir sonraki commit’imizde yer almayacağını belirtebiliriz.

$ **git rm** ornek2.md

**Değişikliklerimizi commit edelim:**

Değişikliklerinizi **git add** ve **git rm** ile staging area'ya aldıktan sonra

**git commit**komutu ile yeni bir versiyon olarak kayıt altına alabilirsiniz.

$ **git commit** -m "1.7 numaralı alt başlık içeriği tamamlandı"

Yukarıdaki komutta yer alan **-m** parametresi ile yaptığınız değişiklikleri özetleyen bir mesajı da commit'inize ekleyebilirsiniz. Eğer birden fazla satırı olan bir commit mesajı gireceksiniz **-m** parametresini kaldırmanız yeterli olacaktır. Bu durumda editör açılır ve bu editöre mesajınızı istediğimiz uzunlukta girebilirsiniz.

**Commit tarihçesi (git log):**

Git projeniz üzerinde çalıştığınız her anda yaptığınız commit işlemlerini kayıt altına almaktadır. Özellikle ekip çalışması söz konusu ise commit işlemleri ile ilgili git tarafından kayıt altına alınan bu bilgiler daha da önem kazanmaktadır.

Git'in commit’leriniz ile ilgili kayıt altına aldığı tarihsel bilgileri görmek için **git log** komutunu kullanıyoruz.

**$ git log**

**Bu komut tüm commit’ler ile ilgili bilgileri, en son commit en üstte olacak şekilde, tarihsel olarak sıralar**. Eğer Terminal pencerenize sığmayacak kadar çok tarihsel kayıt var ise son satırda **:** simgesi yer alacaktır, klavyenizden **SPACE (BOŞLUK**) tuşuna basarak bir sonraki sayfanın listelenmesini, **q** tuşuna basarak da listelemenin sonlandırılmasını sağlayabilirsiniz.

Terminal'de listelenen her commit tarihçesi kaydı, diğer bilgilerin yanı sıra, aşağıdaki temel bilgileri içerir

* Commit'in Hash değeri
* Commit'i gerçekleştiren kişinin adı ve email'i
* Commit tarihi
* Commit mesajı

**Commit Hash** :

Her bir commit'in benzersiz ve tek bir tanımlayıcı değeri vardır. Bu değer git tarafından commit'e dahil olan tüm değişiklikleriniz ve commit'in kendisi ile ilgili bilgiler de kullanılarak otomatik hesaplanır. Genel olarak git'in listelemelerinde ve bazı komutların parametresi olarak bu değerin ilk 7 karakterinin kullanılması yeterlidir. Çünkü bu ilk 7 karakterin de nerdeyse benzersiz ve tekil olduğunu söyleyebiliriz.

git log komutu ile birlikte commit işlemi ile ilgili bilgilendirici çoğu bilgiyi görmekle birlikte parametre olarak **-p** değerini kullanırsanız dosyalarda yapılan değişiklikler de ayrıntılı olarak listelenecektir.

**Remote bir repository’e bağlantı sağlamak:**

Remote bir repository'yi yerel diskinize **git clone** komutu ile indirdiğinizde Git otomatik olarak bu işlemi yapmak için kullandığınız bağlantı bilgilerini hatırlar.

Git bu bilgi'yi varsayılan olarak **origin** adı verilen remote bir repository olarak kayıt altına alır.

Local olan bir respository için ise böyle bir bilgi tutulmaz. Ancak Local bir repository'yi baz alarak yeni bir remote repository oluşturabiliriz. Bunun için **git clone** komutunu kullanabiliriz. Örneğin:

**$ git remote add** git101\_ornek <https://github.com/clarusway/fit101_ornek.git>

Local bir repository'nizi istediğiniz sayıda remote repository ile ilişkilendirebilirsiniz. Yukarıdaki ekran çıktısında sadece bizim oluşturduğumuz **git101\_ornek** isimli remote listeleniyor, birden fazla remote ilişkisi olsaydı hepsi listelenecekti.

**Remote repository’deki verilerin incelenmesi:**

git clone komutu ile remote bir repository'yi yerel diskimize indirdikten sonra

git branch-va komutunu çalıştırdığımızda repository bilgileri listelenecektir.

**Remote Repository bilgileri güncel olmayabilir!**

Git remote repository'ler ile ilgili yerel diskinizde bir takım bilgileri içerir. Ancak Git arka planda otomatik olarak bu bilgileri sizin için belirli aralıklarda güncellemez! Bu işlemin gerçekleşmesi ve sizin diğer takım arkadaşlarınızın yaptığı değişikliklerden haberdar olabilmeniz için Git'e bu bilgileri güncellemesini söylemeniz gerekir.

Git'in remote repository ile ilgili yerel diskinizde tuttuğu bilgileri güncellemesini sağlamak için git fetch komutunu kullanmanız gerekir:

**$ git fetch** git101\_ornek

git fetch komutu yerel diskinizdeki branchlerinizi ve Working Copy'deki dosyalarınızı güncellemez veya değiştirmez. Bu komut ile sadece takım arkadaşlarınızın remote repository'de yayınladıkları değişikliklere ilişkin bilgiler yerel diskinize indirilir. Daha sonra bu değişikliklerden hangilerini hangi local branch'e entegre edeceğinize kendiniz karar verebilirsiniz.

Bu işlemden sonra tekrar git branch -va komutunu

**$ git branch -va**

çalıştırdığımızda gitornek\_101/master isimli remote repositorymizdeki branchlere ilişkin bilgileri de görebiliriz.

Bilgilerini güncellediğimiz git101\_ornek/master isimli branch'de değişiklikler yapmak için öncelikle bu branch'i baz alarak yeni bir local branch oluşturup dosyaların Working Copy alanımıza kopyalanmasını sağlamamız gerekiyor. Bunun için **git checkout** komutunu **--track** parametresi ile kullanıyoruz.

**$ git checkout –track** git101\_ornek/loginsorunu

**git checkout --track** komutu ile aşağıdaki işlemler gerçekleşir

1. Remote branch ile aynı isimde local bir branch oluşturulur
2. Yeni oluşturulan branch aktif hale getirilir
3. --tracking parametresini kullandığımız için yeni oluşan local branch ile remote branch arasında "tracking relationship" adı verilen ve local branch'in hangi remote branch'deki değişiklikleri takip ettiğini gösteren ilişki kurulur.

**Yavaş yavaş git push komutuna geliyoruz:**

Local branch'imizi hazırladığımıza göre gelin şimdi birkaç değişiklik yapalım. Bu değişiklikleri yaptıktan sonra her zamanki gibi önce;

* Değişikliklerimizi staging area'ya alıyoruz ve sonrasında da;
* commit işlemini gerçekleştirerek local repository'de versiyon kontrolüne ilişkin işlemlerimizi bitiriyoruz. Son adım olarak da;
* git push komutu ile localdeki bu değişikliklerimizi **remote branch'de yayınlıyoruz.**

**$ git status**

**Burada değişikliklerimizi modified : dosya1.md örneğindeki gibi görüyoruz. modified olması sadece değişiklik yapıldığını fakat değişikliğin henüz staging areaya alınmadığını ifade ediyor**

**$ git commit -m “klonlanmış local branch’de örnek değişiklik”**

**Burada değişiklikleri commit etmiş olduk.**

**$ git push**

**Burada da değişikliklerimizi remote branch’e göndermiş olduk.**

git push komutu aslında uzun git pushformatındadır. Ancak local branch'imizi oluştururken kullandığımız **--*track*** parametresi sayesinde kurulan "Takip İlişkisi" sayesinde push komutunun uzun hali yerine sade hali olan git push formatında kullanabiliyoruz.

**Remote değişiklikleri entegre etmek:**

Takım arkadaşlarınız kendi değişikliklerini tamamlayıp remote branch'de yayınladıktan sonra siz de bu değişiklikleri inceleyip kendi local branch'inize entegre ederek çalışmanıza devam edebilirsiniz. Ancak remote branch'deki değişiklikleri entegre etmeden önce bu değişikliklere ilişkin bilgileri (dosyaları değil sadece değişikliklere dair Git'de tutulan bilgiler) görmeniz ve incelemeniz gerekir.

**$ git fetch origin**

**Burada değişiklikleri göreceksiniz**

Remote branch'deki değişiklikleri indirmek için git fetch komutunu kullanıyoruz. Git fetch komutuna geçilen *origin* değeri ise *remotes/origin/master* isimli remote branch bağlantısına referans vermek için kullanılır.

*origin* değeri git fetch komutunun bir parçası değil sadece bir parametre. Origin yerine daha önce local branchimiz ile bağlantısını/ilişkisini kurduğumuz herhangi bir remote branch'i gösteren bir değer olabilir.

git fetch komutu ile remote branch'deki değişiklikleri indirdikten sonra ise:

git log komutunu kullanarak bu remote branch'deki değişiklikler ile ilgili bilgileri görebiliriz. (değişiklik tarihi, kimin yaptığı, değişen dosyalar ve commiti sırasında girilen mesaj gibi)

Değişiklikleri inceledikten sonra bunları local branch'inize entegre etmeye karar verdiğimizde ise **git pull** komutunu kullanmamız gerekecek.

Remote branchdeki değişikliklerin bilgilerini indirmek için kullanılan fetch (türkçe anlamı [getirmek](http://www.seslisozluk.net/?word=fetch&lang=tr-en)) ve bu değişiklikleri entegre etmek için kullanılan pull (türkçe anlamı [çekmek](http://www.seslisozluk.net/?word=pull&lang=tr-en)) ifadelerinin birbirine yakın anlamları olduğu için karıştırabilirsiniz. Bu karışıklığın önüne geçmek için yapacağınız en güzel şey **git pull komutunu hiç kullanmamak olacaktır**.

**Git pull komutu aslında arka arkaya iki şey yapmanızı sağlar:**

* Remote branch'deki değişiklikler ile ilgili bilgileri indirmek, yani **git fetch**
* Remote branch'deki değişiklikleri local branch'inize entegre etmek yani **git merge**

**Branch’leri silmek:**

Dana önce oluşturduğumuz **superyeniozellik** isimli branch üzerindeki çalışmamızı tamamlayıp kalite kontrol sürecimizi de işlettikten sonra bu değişiklikleri **master(main)** branch'imize entegre ettiğimizi varsayalım.

Bu entegrasyon sonrasında **superyeniozellik** isimli branch'e ihtiyacımız yok ve artık bu branch'i silebiliriz.

Bu branch'i kendi bilgisayarımızdan silmek için **git branch -d superyeniozellik** komutunu,

**$ git branch -d** superyeniozellik

Remote repository'den silmek için de **git branch -dr superyeniozellik** komutunu kullanabiliriz.

**$ git branch -dr** origin/superyeniozellik

**Önemli not1:** Silmek istediğiniz local branch aktif ise **git branch -d** komutu hata verecektir. Silme işlemi öncesinde sileceğiniz local branch'den farklı bir branch'i **git checkout** komutu ile aktif hale getirmeyi unutmayın.

**Önemli not2:** Remote branch'i **git branch -dr** komutu ile sildiğiniz halde remote repository'ye erişip branchleri kontrol ederseniz **superyeniozellik** isimli branch'in sunucuda hala durduğunuz göreceksiniz. Bunun nedeni **git branch -dr** komutundaki seçeneklerden **r** seçeneğinin **sunucudaki branch'i değil yerel bilgisayarınızda remote branch bilgilerini siler**. Bu değişikliğin sunucuda da geçerli olması için yani sunucudaki branch'i de silmek için **git push origin :superyeniozellik** komutu ile **değişikliği bir anlamda remote repositry'de yayınlamanız gerekiyor.**