

git 101

Git Versiyon Kontrol Sistemine Giriş

Ali Özgür

Twitter | GitHub @aliozgur
<http://aliozgur.net>

İçindekiler

Introduction	1.1
Versiyon Kontrolüne Giriş	1.2
Versiyon Kontrolü Nedir?	1.2.1
Versiyon Kontrolüne Neden İhtiyacımız Var?	1.2.2
Kısa Git Tarihçesi	1.2.3
Git İle Çalışmaya Başlamak	1.2.4
Basit Anlamda Versiyon Kontrolü İş Akışı	1.2.5
Local bir proje oluşturmak	1.2.6
Remote bir proje oluşturmak	1.2.7
Projemiz Üzerinde Çalışmaya Başlayalım	1.2.8
Branching (Dallanma) ve Merging (Birleştirme)	1.3
Branching Çalışma Şeklinizi Değiştirebilir	1.3.1
Branch'ler İle Çalışmak	1.3.2
Değişikliklerinizi Geçici Olarak Kaydetmek -> Git Stash	1.3.3
Local Bir Branch'de Çalışmak	1.3.4
Değişiklikleri Merge Etmek	1.3.5
Branching İş Akışları	1.3.6
Remote Repository'ler	1.4
Remote Bir Repository'ye Bağlantı Sağlamak	1.4.1
Remote Repository'deki Verilerin İncelenmesi	1.4.2
Remote Değişiklikleri Entegre Etmek	1.4.3
Local Bir Branch'i Yayınlamak (Publish)	1.4.4
Branch'leri Silmek	1.4.5
İleri Seviye Komutlar ve İşlemler	1.5
Değişikliklerinizi Geri Almak	1.5.1
Diff İle Farkları İncelemek	1.5.2
Çakışmaları Gidermek	1.5.3
Merge Alternatifi Olarak Rebase Kullanımı	1.5.4
Git Araç ve Servisleri	1.6
Görsel Git İstemcileri	1.6.1

Diff/Merge Araçları	1.6.2
Git Servisleri	1.6.3
Kaynakça ve Referanslar	1.6.4
Git ile Versiyon Kontrolü - Günlük Alıştırmalar ile Tekrar	1.7
1.Gün: İstemci Kurulumları	1.7.1
2.Gün: Yerel Depo Oluşturma	1.7.2
3.Gün: Klonlama	1.7.3
4.Gün: Değişiklikleri Kaydetme	1.7.4
5.Gün: Değişiklikleri Versiyon Kontrolüne Alma	1.7.5
6.Gün: Commit Edilmiş Değişiklikleri İptal Etmek	1.7.6
7.Gün: Commit Edilmiş Değişiklikleri Silmek	1.7.7
8. Gün: Dal Oluşturmak	1.7.8
9.Gün : Değişiklikleri Birleştirme (Merge)	1.7.9
10.Gün: Rebase	1.7.10
11.Gün: Birlikte Çalışma (Collaboration)	1.7.11

Türkçe Git 101

Önsöz

Kitabın basılı halini online olarak da satın alabilirsiniz

Bu kitapta son 4-5 yılda yazılım geliştiricilerin ve yazılım şirketlerinin vazgeçilmez araçlarından biri olan ve benim de bir yazılım geliştirici olarak çok başarılı bulduğum Git Dağıtık Versiyon Kontrol Sistemini (Distributed Version Control System) örnekler ile ele alarak size tanıtmaya çalışacağım.

İster kıdemli bir yazılım geliştirici olun isterseniz yazılım geliştirme işine yeni başlayan bir profesyonel iş görüşmelerinde temel bilgisayar bilimleri, programlama dilleri ve veri tabanları ile ilgili sorulardan sonra ilk 5 soru içinde yer alacak konulardan birisi de Git olacaktır. Özellikle 2009 yılı itibariyle bulut tabanlı bir sistem olarak kullanıma açılan GitHub'ın yıllar içindeki yükselişi ile birlikte GitHub profiliniz, GitHub'da takip ettiğiniz ve katkı yaptığınız projeler işverenler açısından sizinle ilgili önemli ipuçları sağlayan bir bilgi olarak değerlendirilmektedir.

Versiyon kontrol sistemi olarak Git'in yükselişi GitHub'ın yükselişi ile paralellik göstermiştir. 2005 yılında ilk stabil sürümü yayınlanan Git, 2009 ve 2010 yılına kadar sadece Linux ile iç içe olan çevrelerin takip edip kullandığı dağıtık bir versiyon kontrol sistemiydi. Ancak 2009 itibariyle GitHub'ın açık kaynak projeler için ücretsiz Git hizmeti vermeye başlaması, hemen ardından aynı yıllarda Bitbucket'in ve sonrasında GitLab'ın benzer Git servislerin sağlamaya başlaması Git'in daha geniş kitlelere ulaşmasını ve farklı profillerdeki yazılım geliştiriciler tarafından yoğun bir şekilde kullanılmasını sağladı.

Yazılım Geliştirme ile ilgili çoğu konuda olduğu gibi Git ile ilgili kaynaklar da ağırlıklı olarak İngilizce yazılmaktadır. Türkçe bir kaynak kitap hazırlamanın terminoloji anlamında en büyük zorluğu İngilizce terimler için uygun Türkçe karşılıklarını bulmaktır. Ancak bu kitapta İngilizce-Türkçe geçişini birebir yapmayacağız, mümkün olduğu kadar Versiyon Kontrolü ve Git ile ilgili terimlerin İngilizce hallerine yer vereceğiz. Yazılım Geliştirme alanında kullanılan araçların çoğu kendi terminolojisi ve jargonu ile bilinir bu nedenle kullandığımız İngilizce terimleri İngilizce birere kelime olarak değil Git ile çalışmanın terminolojisi ve jargonu olarak düşünebilirsiniz. Bu terimler profesyonel yaşamınızda ekip arkadaşlarınız ile yapacağınız konuşmalarda sıkça duyup kullanacağınız Git'e dair terimler olarak zihninize yerleştirmelisiniz.

Konuların diziliminde ve içeriğin oluşturulmasında kendi kişisel tecrübemin yanı sıra Git'i öğrenme ve kendi ekibime bu bilgileri aktarma aşamasında faydalandığım birçok basılı ve çevrimiçi kaynaktan faydalanılmıştır.

Bu kitapta yer verdiğimiz giriş ve temel seviyedeki konuları iyice kavrayıp kendi projelerinizde uyguladıktan sonra daha ileri seviyede Git öğrenmek isterseniz ücretsiz e-kitap olarak yayınlanan ve her zaman güncel tutulan Pro Git isimli kitaptan faydalanabilirsiniz. Pro Git kitabı Git'in temel kaynaklarından birisidir ve Git ile ilgili yayınlanan çoğu basılı ve çevrimiçi kaynak bu kitaba atıfta bulunmaktadır.

Kitaptaki örneklerimizi Terminal (komut satırı veya command line olarak da tabir edilen) üzerinden Apple OSX ve Windows işletim sistemleri üzerinde oluşturacağız. Bu kaynağın oluşturulduğu sırada kullanılan OSX ve Git sürüm bilgileri şöyleydi

- OSX 10.12.3 (Sierra)
- Git 2.9.3 (Apple Git-75)

Windows tarafında ise sürüm bilgileri şu şekilde

- Windows 10 Pro 64 bit
- Git 2.9.2 (Git for Windows)

Git, OSX'in yanı sıra tüm Linux dağıtımları ve Windows'da da çalışmaktadır. Git komutları kullandığınız işletim sistemine göre değişmez ancak Git kurulumu ve kullanacağımız yardımcı araçlar işletim sistemine göre değişebilir.

Versiyon Kontrolüne Giriş

Bu bölümde aşağıdaki konuları ele alacağız

- Versiyon Kontrolü Nedir
- Versiyon Kontrolüne Neden İhtiyacımız Var
- Git İle Çalışmaya Başlamak
- Basit Anlamda Bir Versiyon Kontrolü İş Akışı
- Local Bir Proje Oluşturmak
- Remote Bir Proje Oluşturmak
- Projemiz Üzerinde Çalışmaya Başlayalım

Versiyon kontrol sistemleri en basit anlamda **dosyalarınızda ki deęişikliklerin tarihçesini takip edip kayıt altında tutan** sistemlerdir. Bu nedenle versiyon kontrol sistemlerini yedekleme veya dięer yazılım geliştirme araçları ile karşılaştırmak doğru olmaz.

Kitabın basılı halini online olarak da satın alabilirsiniz

Versiyon Kontrolüne Neden İhtiyacımız Var?

Versiyon kontrol sistemi kullanmanın bir çok faydası var ve bu bölümde versiyon kontrol sistemi kullanımının bize sağladığı avantajlardan bahsediyoruz.

Uyumlu ekip çalışması

Herhangi bir versiyon kontrol sistemi kullanmadığınızda beraber çalıştığınız diğer kişiler ile aynı dosyalar üzerinde çalışabilmek için muhtemelen herkesin erişimine açık paylaşımlı bir klasör kullanmak zorunda kalacaksınız.

Bu tür bir senaryoda kullanılan yazılımların çoğu değiştirilen dosyaya **kilit** koyar ve başka birisi aynı dosyayı düzenlemek istediğinde

- Kullandığı programa bağlı olarak dosya yazma korumalı olarak salt okunur modda (readonly) açılır veya
- Değişiklikler kaydedilmek istendiğinde hata verir

Bu tür bir çalışma hem çok zahmetli hem de hatalara açıktır. Örneğin bir dosyanın en son geçerli versiyonunun nerede olduğunun takip edilmesi gibi çözüm bulunması gereken sorunlar ile uğraşmak zorunda kalırsınız.

Üzerinde çalıştığınız dosyada sizden önce başkasının değişiklik yapıp yapmadığından haberiniz yoksa hatalı içerik üretme ihtimaliniz vardır.

Versiyon kontrol sistemi kullanıldığında ise ekibinizdeki herkes özgür bir şekilde istediği dosyalar üzerinde güvenli bir şekilde istediği değişikliği yapabilir. Herkes değişikliklerini tamamladıktan sonra da tüm değişiklikler versiyon kontrol sistemi kullanılarak sağlıklı bir şekilde **merge** (*birleştirme*) edilebilir.

Versiyonların düzgün bir şekilde takip edilebilmesi

Üzerinde çalıştığınız bir dosyanın veya bir dizi proje dosyasının zaman içinde farklı versiyonları oluşur ve bu versiyonların kayıt altına alınması gerekir. Bu sorumluluk genelde çok zahmetli ve sıkıcı bir iş ve süreçtir. Aşağıdakine benzer sorular canınızı gereğinden fazla sıkabilir

- Sadece değişen dosyalar mı yoksa bir projedeki tüm dosyaların versiyonları mı kaydedilmeli?
 - Bir sürü dosya içinden sadece değişen dosyaların belirlenmesi zordur
 - Her seferinde dosyaların hepsinin teker teker kaydedilmesi durumunda ise ihtiyaç duyulandan daha fazla disk alanı kullanılır
- Dosyalara verilecek isimler tam bir baş ağrısına dönüşebilir.
 - Personel_Maas.xlsx
 - Personel_Maas1.xlsx
 - Personel_Maas_Ozet.xlsx
 - Personel_Maas_BrutHaricDetay.xlsx şeklinde dosya isimleri üretmek zorunda kalabilirsiniz.
- Belki de canınızı en çok sıkacak şey projenizin iki versiyonu arasında tam olarak ne tür farkların olduğunu sağlıklı bir şekilde bilme şansınız olmaması olacaktır

Versiyon kontrol sistemi kullandığınızda sizin çalıştığınız disk alanında proje dosyalarının sadece bir versiyonu bulunur, bu dosyaların daha önceki halleri versiyon kontrol sisteminin denetimindedir. Bu sayede istediğiniz zaman önceki versiyonlara geri dönebilir, versiyonlar arasındaki farklılıkları rahatlıkla inceleyebilir ve versiyonları kaydederken eklediğiniz ilave bilgileri ve yorumlarınızı rahatlıkla görebilirsiniz.

Önceki Versiyonlara Geri Dönebilme

Dosyalarınızın veya aslında tüm projenizin daha önceki versiyonuna geri dönebilme imkanın size ciddi anlamda özgürlük sağlar; dosyalarınızı ve projenizi istediğiniz gibi değiştirme özgürlüğü. Yaptığınız değişiklikler projenizi çöpe döndürdüyse, geliştirdiğiniz bir işlev tam istediğiniz gibi olmadıysa veya müşteriniz veya patronunuz geliştirdiğiniz bir işlevi artık istemediğine karar verirse projenizin önceki temiz haline çok hızlı ve rahat bir şekilde dönebilirsiniz.

Dosyalarınızın neden değiştiğini anlama

Versiyon kontrol sistemleri değişikliklerinizi tamamlayıp **commit** etmek istediğinizde **comment** adı verilen açıklamalar girmenizi isterler. Bu comment'ler sayesinde projenizin herhangi bir versiyonundaki değişikliklerin nedenlerini de kayıt altına alıp ihtiyaç halinde geri dönüp inceleyebilirsiniz.

Git'de commit işlemi yapılırken comment (yorum metni) girilmesi zorunludur

Yedekleme

Git gibi dağıtık versiyon kontrol (DVCS) sistemlerinin yan etki olarak sağladığı faydalardan birisi de yedeklemedir. Git sayesinde aynı projede çalışan herkesin kendi bilgisayarında projenin tam bir tarihçesi tutulur. Merkezi versiyon kontrol sistemi sunucusunda bir sorun oluştuğunda takımdaki herhangi birinin kendi diskindeki projeyi sunucuya geri yüklemesi yeterlidir. Diğerleri de kendi bilgisayarlarındaki proje dosyalarını geri yüklenen proje dosyaları ile senkronize edebilirler.

Kısa Git Tarihçesi

Git 2005 yılında, başta Linus Torvalds olmak üzere Linux çekirdeğini de kodlayan ekip tarafından Linux kaynak kodunu versiyon kontrolü altında tutmak ve kendi iş akışlarını düzenlemek için geliştirilmiştir

Linux'un kaynak kodu 1991-2002 yılları arasındaki dönemde manuel olarak dosyaların paylaşılması şeklinde yönetiliyordu. 2002 yılında Linux geliştiricileri normalde ücretli olan ancak açık kaynak projeler için ücretsiz lisanslama modeli sunan BitKeeper isimli dağıtık versiyon kontrol sistemini kullanmaya başladılar. 2005 yılında BitKeeper'ın ücretsiz sağladığı lisansı geri çekmesi üzerine Linus Torvalds ve Linux ekibi kendi dağıtık versiyon kontrol sistemini geliştirmeye karar verdiler.

Linux ekibi BitKeeper ile olan deneyimlerini de dikkate alarak öncelikli olarak aşağıdaki kriterleri sağlayan kendi yazılımlarını geliştirmeye başladılar

- Hızlı
- Kullanımı kolay
- Lineer olmayan geliştirme iş akışına uygun (branching)
- Tamamen dağıtık
- Büyük projeleri destekleyebilecek

2005 yılından bugüne Git gelişmeye devam ediyor. Git'e yeni eklenen özelliklere rağmen Git bugün bile yukarıda bahsettiğim öncelikli kriterlerden taviz vermeden milyonlarca yazılım geliştiricinin hayatını kolaylaştırmaya devam ediyor.

Git İle Çalışmaya Başlamak

Komut satırı mı yoksa görsel arayüz mü?

Git ile çalışmak için git'in kendi **komut satırı arayüzünü** (Git Command Line Interface) veya görsel kullanıcı arayüzü olan masaüstü uygulamalar (SourceTree, Tortoise Git, Tower veya GitHub) kullanabilirsiniz.

Git ile çalışırken görsel arayüzü olan bir uygulama kullanmanız üretkenliğinizi arttırıp Git'in çok sayıdaki karmaşık komutuna daha hızlı ve kolay erişmenizi sağlar. Diğer yandan Git'in komut satırı arayüzünü kullanmanız Git ile ilgili daha ayrıntılı bilgilenmenizi ve 3. parti uygulamalara bağımlı kalmadan Git ile çalışabilmenizi sağlar.

Git komutlarını komut satırında öğrendikten sonra günlük çalışmanızda görsel arayüzü olan bir uygulamayı mutlaka kullanmanızı öneriyorum.

Kurulum

Git'in kurulumu hem Windows hem de Mac OS X için oldukça kolay bir işlemdir. Her iki işletim sistemi için tek tıkla kurulum yapmanızı sağlayan kurulum sihirbazları vardır.

Windows

İşletim sisteminiz Windows ise git ile çalışmak için "msysgit" paketini kullanabilirsiniz.

msysgit paketini kurmak için <http://msysgit.github.io/> adresinden kurulum uygulamasını indirip çalıştırmalısınız. Kurulum adımları sırasında karşınıza çıkacak olan ekranlarda varsayılan ayarları seçili olarak bırakarak kurulumunuzu tamamlayabilirsiniz.

Kurulum tamamlandıktan sonra Windows Başlangıç menüsünden *Git* klasörü altındaki **Git Bash** uygulamasını çalıştırıp Git'in komut satırı arayüzünü kullanmaya başlayabilirsiniz.

Git'in kurulumunun sorunsuz gerçekleştiğini teyid etmek için **Git Bash**'i açıp **git --version** komutunu yazın. Bu komut ekrana Git'in versiyon bilgisini basar. Eğer hata alırsanız msysgit ana sayfasından sorunun giderilmesi için ne yapmanız gerektiğini öğrenebilirsiniz.

Mac OS X

İşletim sisteminiz Mac OS X ise Git kurulumu için iki yöntem kullanabilirsiniz.

- Apple'ın geliştirici araçlarını kurarak (XCode) Apple tarafından sağlanan Git dağıtımını kurabilirsiniz
- [Git OS X Installer](#) paketini indirip Git'i kurabilirsiniz.

Git kurulumunu tamamladıktan sonra Applications klasörü altındaki Terminal.app uygulamasını çalıştırın.

Spotlight'a *terminal* yazarak da Terminal.app uygulamasını bulup çalıştırabilirsiniz

Kurulumunuzu denetlemek için komut satırında **git --version** komutunu çalıştırın. Bu komut Git'in versiyonunu ekrana basar. Herhangi bir hata almanız durumunda kurulum yönteminize göre ilgili kaynakları araştırmanız gerekebilir.

Git Konfigürasyonu

Git'i kurduğumuza göre artık Git ile çalışmak için bazı ayarlar yapabiliriz. Bu ayarlar için Git bize **git config** isimli bir araç/komut sunar. Git ayarlarını bir defa yapmanız yeterli olacaktır.

Bu ayarları istediğiniz zaman değiştirebilirsiniz.

Git ayarlarınız aşağıda belirtilen üç konumda kaydedilir ve hiyerarşik olarak bu konumlardan yüklenir

1. Seviye (/etc/gitconfig dosyası) : Tüm kullanıcı ve projeler için geçerli olan ayarlar bu dosyada kaydedilir. **git config** komutunu **--system** seçeneği ile çalıştırırsanız ayarlar bu dosyada kaydedilecek ve bu dosyadan okunacaktır
2. Seviye (/ .gitconfig dosyası) : Sadece sizin kullanıcınız için tanımlanan ayarların kaydedildiği dosyadır. **git config** komutunu **--global** seçeneği ile çalıştırırsanız ayarlar bu dosyaya kaydedilecek ve bu dosyadan okunacaktır
3. Seviye : Proje klasörünüzün (projenizin Git ile versiyon kontrolüne alınmış olması gerekiyor) altında yer alan **.git/config** dosyasında ise proje bazındaki git ayarlarınız yer alır.

Git, ayarlarınızın değerini belirlemek için bu üç konumdaki dosyaları 3. seviye, 2. seviye ve 1. seviye sıralaması ile hiyerarşik olarak okur. Belirli bir ayar'a ilişkin değere ilk hangi seviyede rastlandıysa o seviyedeki değer dikkate alınır diğer seviyelerdeki değerler dikkate alınmaz.

Windows'da global (**git config --global** komutu) git ayarlarınız Windows'un \$HOME klasörü altında yer alan (genellikle C:\Documents and Settings\$USER) **.config** dosyasında yer alır. Proje seviyesindeki ayarlarınız ise OS X'de olduğu gibi **[Projenizin Ana Klasörü].git\config** dosyasında kayıt altına alınır.

Kullanıcı adınızı ve email bilgisi

Git ayarlarından en önemli olanları kullanıcı adınız ve email adresinizdir. Git, ayar olarak tanımladığınız değerleri **commit** vb işlemlerde otomatik olarak kullanır. Bu ayarların değerini belirlemek için komut satırında aşağıdaki komutları çalıştırıyoruz

```
git config --global user.name "ali özgür"
git config --global user.email "ali.ozgur@example.com"
```

Yukarıdaki komutlarda

- **--global** seçeneği ile Git'e global ayarları düzenlediğinizi söylüyoruz
- **user.name** (ve **user.email**) ile değerini değiştirmek istediğiniz ayarın anahtar'ını belirtiyoruz
- Ardından da çift tırnak içinde ilgili ayarın değerini giriyoruz

Bu ayarları **--global** ibaresi ile tüm projelerinizde geçerli olacak şekilde yaptık, proje seviyesinde bu ayarları yapmak için komut satırında (terminal'de) projenizin klasörüne konumlanıp **git config user.name "ali özgür"** komutu ile **--global** seçeneğini kullanmadan yapabilirsiniz.

Kendi yaptığımız veya kurulum ile hazır gelen ayarların değerlerini görmek için aşağıdaki komutları kullanabiliriz.

- Global seviyede tüm ayarları listelemek için
git config --global -l
- Global seviyede tek bir ayar'ın değerini (örneğimizde user.name anahtarına sahip ayar) görmek için ise
git config --global user.name

İPUCU

Git'in komutları ve bu komutların seçenek ve parametreleri ile ilgili yardım almak istediğinizde

- **git [komut adı] --help** (örneğin: **git init --help**)
- **git help [komut adı]** (örneğin: **git help init**)

komutlarını kullanabilirsiniz.

Editör ayarı

Git'in bazı komutları sizden interaktif olarak yorum veya bilgi girmenizi ister. Bu tür durumlar için Git'in hangi metin düzenleme uygulamasını kullanacağını ayarlayabilirsiniz. Git varsayılan olarak [Vi](#) veya [Vim](#)) kullanır. Ancak bu editörlerin kullanımı başlangıç seviyesindeki kullanıcılar için zor olabilir. Ben, Vi veya Vim ile karşılaştırıldığında kullanımının daha kolay olduğunu düşündüğüm [GNU Midnight Commander \(MC\)](#) kullanmanızı öneriyorum.

Midnight Commander'i Mac OS X'e [Homebrew](#) kullanarak

```
brew install midnight-commander
```

komutu ile kurabilirsiniz.

Midnight Commander veya Git'i destekleyen editör kurulumunu tamamladıktan sonra

```
git config --global core.editor mcedit
```

ile Git'in kullanacağı editör ayarınızı yapabilirsiniz.

Diff aracı ayarları

Diff kavramını ilerleyen bölümlerimizde daha ayrıntılı ele alacağız, ancak kısaca değinmek gerekirse

Bir dosyanın Tx anındaki içeriği ile Ty anındaki içeriğinin arasındaki farkları tespit etme ve gösterme işlemidir. İngilizcede **difference** (fark) kelimesinin kısaltması olan **diff** şeklide kullanılır.

Bu işlemi göz ile yapmak zorunda kalmadan dosyalar ve/veya klasörler arasındaki farkları tespit etmek ve görselleştirmek için kullanılan araçlara genel olarak Diff Araçları ismi verilir.

Ben, Mac OS X üzerinde ücretsiz bir uygulama olan **SourceGear DiffMerge** kullanmayı tercih ediyorum. Git'in diff aracı olarak SourceGear DiffMerge'i kullanmasını sağlamak için

```
git config --global merge.tool diffmerge
```

komutu ile ayar yapabilirsiniz. DiffmMerge'in OS X'de tam olarak ayarlarının nasıl yapılacağını öğrenmek için [bu linkte](#) göz atabilirsiniz.

Windows'da ise yine ücretsiz bir uygulama olan [WinMerge](#) veya ücretli bir uygulama olan [Araxis Merge](#)'i kullanılabilir. Bu araçların Git ayarlarının nasıl yapılacağını yardım dokümanlarından faydalanarak öğrenebilirsiniz.

Basit Anlamda Versiyon Kontrolü İş Akışı

Git'in derinliklerine dalmadan önce gelin basit bir versiyon kontrol iş akışına adım adım göz atalım.

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.

Yukarıda 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 ağınızda bir Git sunucusunda versiyon kontrolü altında tutuluyorsa projeyi kendi bilgisayarınıza **git clone** komutu ile indirebilirsiniz.

Projeniz için yukarıdaki yöntemlerden biri ile **repository** oluşturduktan sonra aşağıdaki basit akışı kullanarak değişikliklerinizi yapmaya başlayabilirsiniz

1. Projenizin repository'sini oluşturduktan sonra dosyalarınız üzerinde istediğiniz değişiklikleri istediğiniz uygulamayı kullanarak yapabilirsiniz. Bu aşamada yaptığınız değişiklikleri versiyon kontrolü için birebir ve doğrudan takip etmenize gerek yoktur.
2. Yaptığınız değişiklikler istediğiniz bir noktaya ulaştığında veya bir özellik veya sorun giderme düzenlemesi ile ilgili çalışmanız tamamlandığında versiyon kontrolü bakış açısı ile değişikliklerinizi değerlendirmeniz gerekir. Bu aşamada değişikliklerinizi **commit** adı verilen bir bütünü olarak tarif etmelisiniz. Böylece projenizin yeni bir versiyonunu oluşturma işleminin ilk adımını tamamlamış olacaksınız.
3. Fakat, commit işlemi öncesinde dosyalarınızda yaptığınız değişikliklerin bir özetini görmek isteyebilirsiniz. **git status** komutu ile hangi dosyaları değiştirdiğinizi, sildiğinizi veya hangi dosyaları eklediğinizi kolayca görebilirsiniz.
4. 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 alırsınız.

Dosyaların içeriğinin değiştirilmiş olması, silinmesi veya yeni dosya eklenmesi bu dosyaların otomatik olarak **staging area**'ya eklenmesini sağlamaz. Bu işlemi ilgili dosyaları seçerek sizin yapmanız gerekir.

5. Dosyalarınızı **staging area**'ya ekledikten sonra şimdi **commit** işlemine hazırsınız.

Commit işlemi ile dosyalarınızdaki değişiklikler yeni bir versiyon olarak Git'de kayıt altına alınır.

6. 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.
7. 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.

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'ler ise 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.

Working copy: Projenizin ana klasörüne *Working Copy* veya *Working Directory* ismi verilir. Bu klasörde projenizde yer alan dosyaların ve klasörlerin bir kopyası bulunur. Versiyon kontrol sistemine projenizin herhangi bir versiyonunu Working Copy'nize kopyalamasını söyleyebilirsiniz, ancak bir anda Working Copy'nizde projenizin sadece bir versiyonu yer alır.

Versiyon kontrolü altına almak istemediğimiz dosyalar

Tüm geliştirme ortamları ve işletim sistemlerinde kullandığımız araçlar tarafından ara bir ürün olarak üretilen ve aslında doğrudan versiyon kontrolü altına almak istemediğimiz dosya veya klasörler olacaktır. Örneğin Mac OS X'in otomatik olarak ürettiği gizli **DS_Store** isimli klasör veya C++ derleyicileri tarafından üretilen **.o** uzantılı **obj** dosyaları gibi. Hangi dosyaların versiyon kontrolü altında tutulacağına ve hangilerinin göz ardı edileceğine Git otomatik olarak karar vermez, bu kararı sizin vermeniz gerekir.

Kullandığınız geliştirme araçlarına bağlı olarak hangi dosyaların göz ardı edilebileceği ile ilgili GitHub'ın yayınladığı [derlemeye](#) göz atabilirsiniz.

Versiyon kontrolü altına almak istemediğiniz dosya ve klasörleri tanımlamak için proje klasörüne eklenen `.gitignore` dosyası kullanılır. Bu dosyaya göz ardı etmek istediğiniz dosya ve klasörlerin tespit edilebilmesi için doğrudan isimler veya basit kurallar ekleriz. Projelerinizi versiyon kontrolü altına aldıktan sonra ilk iş olarak GitHub'ın yayınladığı derlemeyi veya kendi deneyiminiz ve bilginiz ile karar vereceğiniz dosya ve klasörleri `.gitignore` dosyasına ekleyiniz. Projenizin ilerleyen aşamalarında bu işlemi yapmanız biraz daha zahmetli olacaktır.

Şimdi gelin `.gitignore` dosyasında kuralları nasıl tanımlayabileceğimize bir göz atalım

```
*.[oa]
.~
```

İlk satırda `o` veya `a` uzantısı ile biten dosyaların versiyon kontrolü dışında tutulması için bir kural tanımlıyoruz. İkinci satırda ise `~` karakteri ile biten (çoğu metin düzenleme uygulaması geçici dosyaları `~` ile biten dosyalar olarak otomatik oluşturur) dosyaların versiyon kontrolü haricinde tutulması için kural tanımlıyoruz.

`.gitignore` dosyasında tanımlama yaparken aşağıdaki kurallar geçerlidir

- Boş satırlar veya `#` ile başlayan satırlarda yaptığınız tanımlamalar Git tarafından dikkate alınmaz.
- `*`, `?`, `[]`, `{ }`, `!` ve `\` gibi karakterler kullanılarak oluşturulan ve [globbing patterns](#) adı verilen tanımlayıcılar kullanabilirsiniz
- Klasörleri belirtmek için `/` karakteri kullanılır. Örneğin `/projemde/versiyon/kontrolu/istemedigim/bir/klasor/` şeklinde bir tanım yaptığımızda ilgili klasör ve altındaki tüm dosyalar Git tarafından göz ardı edilir.
- Tanımladığınız bir kuralın tersini `!` simgesi ile tanımlarız. Örneğin `!/projemin/kaynak/kodu/` şeklinde bir tanım yaptığımızda bu klasör dışındaki tüm klasör ve dosyalar Git tarafından göz ardı edilecektir.

İlk commit'imiz

Projemizi versiyon kontrolüne alıp göz ardı edilmesini istediğimiz klasör ve dosyaları da belirledikten sonra aşağıdaki komutlar ile ilk commit işlemimizi yapabiliriz

```
$ git add -A
$ git commit -m "İlk commit işlemimizi yaptık"
```

Bu komutların ne işe yaradığına sonraki bölümlerde değineceğiz, şimdilik

- İlk komutun tüm proje dosyalarının Staging Area'ya eklenmesi için,

- İkinci komutun ise dosyalarımızın bir açıklama ile commit edilmesi için kullanıldığını söylemek ile yetinelim.

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

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** (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 ve erişim için gerekli 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

```
Alis-MacBook-Pro-2:~ aliozgur$ cd Projects
Alis-MacBook-Pro-2:Projects aliozgur$ mkdir git101_kitap
Alis-MacBook-Pro-2:Projects aliozgur$ cd git101_kitap
Alis-MacBook-Pro-2:git101_kitap aliozgur$
```

Yukarıdaki ekran görüntüsünde yer alan ilk **cd** komutu ile proje klasörümün içinde yer alacağı ana klasör olan **Projects** klasörüne konumlanıyoruz. İkinci komut olan **mkdir** ile proje klasörümüz olan **git101_kitap** klasörünü oluşturuyoruz. Üçüncü komutumuz ile de yeni oluşturduğumuz **git101_kitap** klasörüne konumlanıyoruz.

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

```
Alis-MacBook-Pro-2:git101_kitap aliozgur$ git clone https://github.com/aliozgur/git101_book.git
Cloning into 'git101_book'...
remote: Counting objects: 107, done.
remote: Total 107 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (107/107), 228.68 KiB | 104.00 KiB/s, done.
Resolving deltas: 100% (51/51), done.
Checking connectivity... done.
Alis-MacBook-Pro-2:git101_kitap aliozgur$
```

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>

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. İlerleyen bölümlerde ayrıntılı olarak ele alacağımız **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 eklememiş dosyalardır. Bu dosyalardaki değişiklikler siz dosyaları git'e eklememiş 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.

Versiyon Kontrolünün Altın Kuralları

#1 Sadece Birbiri İle Alakalı Değişiklikleri Commit Edin

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.

```
Alis-MacBook-Pro-2:Git101 aliozgur$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   bolum_1_-_baslangic/projeniz_uzerinde_calismaya_baslayalim.md

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

    bolum_1_-_baslangic/03_gitstatus.png

Alis-MacBook-Pro-2:Git101 aliozgur$
```

Yukarıdaki terminal ekran görüntüsünde de görebileceğiniz gibi git oldukça ayrıntılı durum bilgisi sunmaktadır. **git status** komutu 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 eklemmediğimiz dosyalardır. Bu dosyalar bir önceki grubun içine eklemmediğimiz sürece bir sonraki commit'e dahil olmayacaklardır
- Untracked files (Versiyon takibinde olmayan dosyalar): Bu gruptaki dosyalar ise henüz versiyon kontrolü altına almadığımız dosyalardır.

"git add" ve "git rm" komutları

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.

```
$ 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 1.3 numaralı bölümde ayarladığınız editör açılır ve bu editöre mesajınızı istediğimiz uzunlukta girebilirsiniz.

Versiyon Kontrolünün Altın Kuralları

#2 Anlamlı Commit Mesajları

Commit işlemi sırasında yazacağınız bilgilendirici bir mesaj hem ekibinizdeki diğer kişilerin hem de daha sonra kendinizin yapılan değişikliği daha rahat ve hızlı anlamasını sağlayacaktır. Mesajınıza kısa bir özet satırı yazdıktan sonra bir sonraki satırda da değişikliğin nedeni ve içeriği hakkında bilgi verebilirsiniz.

İyi Bir Commit Nasıl Olmalı?

1. Commit'inizde sadece kavramsal olarak ilişkili değişiklikleri içermeye özen göstermelisiniz. Zaman zaman iki farklı konu veya sorun ile ilgili aynı anda veya çok kısa aralıklarla değişimli olarak çalışmak zorunda kalabilirsiniz. Bu şekilde yapılan bir çalışma sonrasında commit zamanı geldiğinde mümkün ise iki konu ile ilgili değişikliklerinizi bir defada commit etmek yerine iki defada ayrı ayrı commit edin. Bu çok zor oluyorsa kısa yoldan bir anda tek bir değişikliğe odaklanmayı da düşünebilirsiniz.
2. Tamamlanmamış değişikliklerinizi kesinlikle commit etmemeye özen gösterin. Eğer zaman zaman değişikliklerinizi kayıt altına almak istiyorsanız commit işlemi yerine Git'in **Stash** özelliğini/komutunu kullanabilirsiniz.
3. Test edilmemiş değişiklikleri commit etmemeye özen gösterin. Bu öneri aslında bir önceki önerimiz ile pratikte aynı anlama geliyor
4. Commit'leriniz kısa ve açıklayıcı mesajlar içermeli.
5. Son olarak da sık sık commit işlemi yapmayı alışkanlık haline getirmenizi önerebiliriz. Bu alışkanlık ile birlikte yukarıdaki maddeleri de yerine getirebilerseniz iş yapma şekliniz ve konsantrasyonunuz da olumlu yönde etkilenecektir.

Commit Tarihçesi

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. 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 listeleme sonlandırılmasını sağlayabilirsiniz.

```
Alis-MacBook-Pro-2:Git101 aliozgur$ git log
commit 0980cdaff72c66c24bdead247dc76259f1446366
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 17 23:03:25 2014 +0300
```

1.7 numaralı alt başlığa yeni içerik eklendi.

```
commit e0cb99687dcc8d921831268d74492e9fb3be9af0
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sat Aug 16 23:05:15 2014 +0300
```

1.7 numaralı alt başlığın ilk kısmı yazıldı

```
commit b5dc61028b8d8f3ac4209d96ffd0ebec06f0d144
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sat Aug 16 22:36:27 2014 +0300
```

1.6 numaralı alt başlık tamamlandı

```
commit b0c63ed0911e33b20d73865ec91cf287af1c30f8
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 10 23:47:18 2014 +0300
```

1.6 numaralı alt başlığa eklemeler yapıldı

```
commit 4f8d8d7b6ac074f9ef729b814a5e99f9183c0e1b
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 10 23:33:22 2014 +0300
```

1.6 başlığı eklendi.

```
commit a7d4017f055405392c8965d87afff0d95d775f98
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 10 23:14:57 2014 +0300
```

1.5 numaralı alt başlığı içeriği tamamlandı.

```
commit a3155df0b28b20038c91d75ea8521847f0acceed
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 10 21:54:43 2014 +0300
;
```

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 değer 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.

```
Alis-MacBook-Pro-2:Git101 aliozgur$ git log -p
commit 0980cdaff72c66c24bdead247dc76259f1446366
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sun Aug 17 23:03:25 2014 +0300

    1.7 anumaralı alt başlığa yeni içerik eklendi.

diff --git a/README.md b/README.md
index 52279a5..22520a2 100644
--- a/README.md
+++ b/README.md
@@ -9,10 +9,11 @@ Yazılım Geliştirme ile ilgili çoğu konuda olduğu gibi maalesef Git ile ilg
 * İngilizce->Türkçe geçişinde terimlerin anlamını yitirmesine neden olmadan
 * Uluslar arası ekipler ve özellikle açık kaynak projelerde ortak dil İngilizce olduğu için Git ile ilgili terminolojiyle t

-
+<!--
# Konuların Dizilimi

Konuların diziliminde ağırlıklı olarak [Learn Version Control with Git](http://www.git-tower.com/learn/ebook/command-line/
+-->

# Örnekler

diff --git a/bolum_1_-_baslangic/03_gitstatus.png b/bolum_1_-_baslangic/03_gitstatus.png
new file mode 100644
index 0000000..03932d5
Binary files /dev/null and b/bolum_1_-_baslangic/03_gitstatus.png differ
diff --git a/bolum_1_-_baslangic/README.md b/bolum_1_-_baslangic/README.md
index d939116..dc2bb2a 100644
--- a/bolum_1_-_baslangic/README.md
+++ b/bolum_1_-_baslangic/README.md
@@ -1,7 +1,8 @@
# Versiyon Kontrolüne Giriş

Bu bölümde aşağıdaki konuları ele alacağız
-* Versiyon kontrolü nedir?
+
+* Versiyon Kontrolü Nedir?
+ * Versiyon Kontrolüne Neden İhtiyacımız Var??
+ * Git İle Çalışmaya Başlamak
```

Kitabımızın ilerleyen bölümlerinde **git log -p** komutu ile gördüğümüz bilgileri nasıl yorumlayacağımızı ayrıntılı olarak ele alacağız.

Branching (Dallanma) ve Merging (Birleřtirme)

Bu bölümümüzde ařağıdaki konuları ele alacağız

- Branching Çalışma Şeklinizi Değıřtirebilir
- Branch'ler İle Çalışmak
- Değıřikliklerinizi Geçici Olarak Kaydetmek -> Git Stash
- Basit Bir Branching Akışı
- Değıřiklikleri Merge Etmek
- Farklı Branching İş Akışları

Branching Çalışma Şeklinizi Değiştirebilir

Bazı araçların sağladığı imkanlar günlük iş yapma şeklimizi çok derinden etkileyip, yaptığımız işe daha farklı bakabilmemizi sağlar. Git'in **branching** yaklaşımı (Türkçeye **dallanma** olarak da çevirebiliriz) da bahsettiğim bu dönüştürücü etkiye sahip araçlardan birisidir. Branching konusundaki hakimiyetimizin artması ve sağlamlaşması ile birlikte daha farklı iş yapmaya başlayıp daha iyi birer yazılım geliştirici olabilirsiniz.

Branching denilen yöntem aslında Git dışındaki diğer versiyon kontrol sistemlerinde de öteden beri kullanılmakta ve yazılım geliştiricilerin hayatını önemli derecede kolaylaştırmaktadır. Ancak, Git'deki branching yaklaşımı kullanım kolaylığı ve yüksek performansı nedeniyle kendine has olduğunu da söylemeliyiz.

Öyleyse gelin şimdi yavaş yavaş branching'in (dallanma) ne olduğunu anlayalım.

Birden Fazla Bağlamda Çalışmak

Daha önceki bölümün son alt başlığında (git commit) zaman zaman bireysel olarak kısa zaman dilimlerinde aynı projenin farklı özellikleri ile ilgili değişiklikler yapılması gerekebileceğinden bahsetmiştik. Büyük projelerde ise bu durum kişisel bir tercih olmaktan çıkıp iş bölümü/uzmanlık gibi kriterlere bağlı olarak proje/ürün yönetiminin önemli bir parçası halinde ele alınır. Örneğin 5 kişilik bir ekibin her bir üyesi aynı yazılımın farklı özellikleri ile ilgili çalışabilir veya iki farklı kişi aynı özelliğin farklı şekillerde nasıl geliştirilebileceği ile ilgili deneysel çalışma yapıyor olabilirler. Bahsettiğim tüm bu alternatif senaryolar aslında kendi yaşam döngüleri olabilen, çoğu zaman kısa veya uzun süreli eş zamanlı ilerleyen farklı birer **bağlama** denk gelir.

Pratikte üzerinde çalıştığınız projenin/yazılımın her zaman son stabil durumu yansıtan **ana** bir bağlamı ve X numaralı hata bildiriminin düzeltilmesi, yeni bir Y özelliği üzerinde yapılan çalışma veya deneysel bir özellik ile ilgili yapılan çalışma gibi birden fazla yan bağlamı olacaktır.

Branching olmasa da olur mu?

Net olarak birbirinden ayrılmış farklı bağlamlar oluşturmak için branching benzeri araçlar olmasaydı aşağıdakilere benzer senaryolarda nasıl davranacağımız konusunda sıkıntılar yaşayacaktık

- Örneğin müşteriniz veya yöneticiniz iki alternatif sayfa tasarımından birincisini değil de

ikincisini beğendi ve bu arada siz de sayfa tasarımı dışında birkaç tane bug fix ve birkaç tane de dokümantasyon değişikliğini farklı zamanlarda tamamladınız. Bu durumda müşterinizin beğendiği ikinci tasarımı diğer tüm düzenlemeleri kaybetmeden nasıl devreye alacaktınız?

- Üzerinde çalıştığınız alışveriş sitesi için özel olarak geliştirdiğiniz Sepet modülü yerine 3. parti bir modül kullanılması kararı alındı ve sizin de kendi modülünüzü ana yazılımdan sökmeniz istendi. Bu durumda sökmeniz gereken modül kodunu tespit edip diğer modülleri etkilemeden nasıl sökecektiniz?
- Yeni geliştirdiğiniz Beni Haberdar Et işlevi yazılımınızın geri kalan özelliklerinin bir çoğunun değiştirilmesine sebep olmuşken birden Beni Haberdar Et işlevinin saçma ve gereksiz olduğuna karar verilseydi bu işlevi aradan geçen zamanda yazılımın farklı yerlerinde yapılan diğer değişikliklerden izole ederek nasıl çöpe atacaktınız?

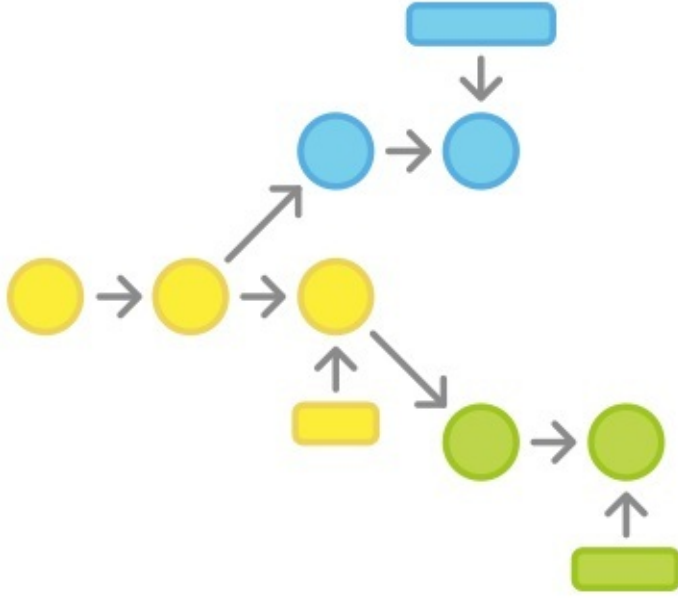
Birden fazla konu ile ilgili değişikliklerin tamamını tek bir bağlam ile yönetmeye çalışırsanız işler hızla sarpa saracaktır. Bu karmaşanın önüne geçmek için her bir değişiklik için projenizin tamamının farklı klasörlere kopyalamayı deneyebilirsiniz. Ancak bu durumda

- Bu klasörler versiyon kontrolünde olmadığı için ekibin geri kalanı ile iş birliği yapmanız çok zorlaşacak
- Farklı değişiklikleri entegre etmek çok zor ve hataya açık bir işlem olacak

Uzun lafın kısıası projenizdeki değişiklikleri profesyonel bir yaklaşımla ele almak istiyorsanız farklı bağlamlarda çalışmak ve bu bağlamları düzgün yönetmek için bir yol bulmanız gerekiyor.

Neyse ki branching var

Branching bir önceki bölümde değindiğimiz tüm sorunların önüne geçmek için kullanabileceğimiz bir araç ve yaklaşımdır. Branching ile farklı bağlamları birbirinde kolayca izole ederek her birini kolayca ve ayrı ayrı yönetebilirsiniz.



Görsel : Atlassian Git Workflows sayfasından alıntı

Herhangi bir anda yaptığınız değişiklikler sadece aktif olarak üzerinde çalıştığınız branch'e (dal) yansıyacak diğer branch'ler bu değişikliklerden etkilenmeyecektir. Böylece aynı anda birden fazla branch üzerinde özgürce çalışabilirsiniz ve en önemlisi de bu çalışmalarınızdan bir kısmının çöpe dönmesinden çekinmeden denemelerinizi yapabilirsiniz.

Versiyon Kontrolünün Altın Kuralları

#3 Branch'leri Bol Bol Kullanın

Branch'ler git'in en güçlü özelliklerinden birisidir. Hızlı ve kullanımı kolay branching mekanizması git'in tasarımında ilk gününden itibaren ciddi bir gereksinim olarak ele alınmıştır. Branch'ler farklı bağlamlarda çalışmaktan kaynaklanabilecek karmaşanın önüne geçmek için biçilmiş kaftandır. Branch'leri bug fix'ler, yeni özellikler üzerinde çalışmak veya deneysel özellikleri geliştirmek için bol bol kullanın.

Branch'ler İle Çalışmak

Git'de branch kullanımı tercihe bağlı değildir, aslında farkında olmasanız bile projeniz üzerinde çalışırken her zaman aktif tek bir branch üzerinde çalışırsınız. Git'de projenizi ilk oluşturduğunuzda Git varsayılan olarak sizin için **master** adı verilen bir branch oluşturur ve siz bu branch üzerinde çalışmaya başlarsınız.

Gelin şimdi **git branch** komutunun basit kullanımı ile ilgili birkaç örnek görelim.

git branch deneme komutunu çalıştırdığınızda git sizin için projenizdeki dosyaların o anki halini barındıran **deneme** isimli bir branch oluşturur.

Git **git branch** komutu ile oluşturduğunuz yeni branch'i otomatik olarak aktif hale getirmez.

Branch'inizi oluşturduktan sonra **git branch** komutunu çalıştırdığınızda git size projeniz için oluşturduğunuz tüm branch'leri listeler ve aktif olan branch'i başına da * simgesi olacak şekilde gösterir.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git branch deneme
Alis-MacBook-Pro-2:git101 aliozgur$ git branch
deneme
* master
test
Alis-MacBook-Pro-2:git101 aliozgur$
```

git status komutunu çalıştırdığınızda da aktif olan branch "On branch" ifadesi ile

```
Alis-MacBook-Pro-2:git101 aliozgur$ git status
On branch master
nothing to commit, working directory clean
Alis-MacBook-Pro-2:git101 aliozgur$
```

gösterilir

Branch'leriniz ile ilgili daha fazla ayrıntı görmek için ise **git branch** komutunu **-v** parametresi ile çalıştırabilirsiniz.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git branch -v
deneme de18052 İlk commit işlemi
* master de18052 İlk commit işlemi
test de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

Yeni oluşturduğumuz branch ile çalışmaya başlamadan önce gelin bir defa daha **git status** komutu ile projemizin ne durumda olduğuna bakalım.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   dosya2.md

no changes added to commit (use "git add" and/or "git commit -a")
Alis-MacBook-Pro-2:git101 aliozgur$
```

Yukarıdaki ekran görüntüsünde de gördüğümüz üzere aktif olan **master** branch'imizde *dosya2.md* isimli dosyamızda henüz commit etmediğiniz bir değişiklik var. Bu dosyadaki değişikliğin yeni eklediğimiz branch'de yer almasını istemediğimizi ve henüz tam anlamıyla bitirilmediğini varsayalım. Bu durumda dosyadaki değişikliği commit mi etmeliyiz yoksa tamamen göz ardı mı etmeliyiz?

Versiyon Kontrolünün Altın Kuralları

#4 Yarım Yamalak Değişiklikleri Asla Commit etmeyin

Tam anlamıyla bitirmediğiniz ve test etmediğiniz bir değişikliği asla commit etmeyin. Üzerinde çalışacağınız değişiklikleri planlarken bu değişiklikleri mümkün olduğunca küçük parçalar halinde ele almaya özen gösterirseniz yaptığınız değişiklikleri kayıt altına almak için henüz tamamlanmamış değişiklikleri commit etmek zorunda kalmazsınız. Buna rağmen ara safhada kayıt altına almak istediğiniz değişiklikler olursa Git'in ****Stash**** özelliğini kullanabilirsiniz.

Değişikliklerinizi Geçici Olarak Kaydetmek -> Git Stash

Commit işlemi ile dosyalarınızda yaptığınız değişiklikler kalıcı olarak repository'de kayıt altına alınır. Ancak günlük çalışmamızda bazen tam olarak bitmeyen değişiklikleri de kayıt altına almak isteyebiliriz. Örneğin bir değişiklik üzerinde çalışırken başka bir konu ile ilgili kritik bir sorun bildirildiğinde yapmakta olduğumuz işi yarım bırakıp yeni soruna odaklanmak zorunda kalabilirsiniz.

Bu gibi durumlarda yeni sorun ile ilgilenmeye başlamak için önceki değişikliklerinizi kaybetmeden yeni ve temiz bir branch oluşturmalsınız. Yarım kalan değişiklikleri kayıt altına almak için **git stash** komutunu kullanmalısınız.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash
Saved working directory and index state WIP on master: de18052 İlk commit işlemi
HEAD is now at de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$ git status
On branch master
nothing to commit, working directory clean
Alis-MacBook-Pro-2:git101 aliozgur$
```

git stash ile üzerinde çalıştığınız ancak henüz commit etmediğiniz değişikliklerin geçici olarak Git tarafından kayıt altına alınmasını ve aktif branch'inizin herhangi bir değişikliğin olmadığı temiz bir duruma getirilmesini sağlarsınız. **git stash** komutunu çalıştırdıktan sonra tekrar **git status** komutunu çalıştırırsanız önceki bölümde commit edilmemiş bir değişiklik olarak görünen *dosya2.md* dosyasındaki değişiklik artık listelenmez çünkü **master** branch'imiz **git stash** sonrası temiz bir duruma geldi.

git stash list komutunu kullanarak aktif branch'inizde geçici olarak kayıt altına aldığınız değişikliklerin listelenmesini sağlayabilirsiniz.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
stash@{2}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

Yukarıda görünen listede en son stash işlemi ile geçici olarak kaydedilen değişiklikler en üstte yer alır. Stash'de yer alan bir değişikliği geri yüklemek istediğinizde iki seçeneğiniz var

- **git stash pop** komutu ile yukarıdaki listenin en üstünde yer alan değişiklik geri yüklenecek ve bu değişiklik listeden silinecek.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
stash@{2}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   dosya1.md
        modified:   dosya3.md

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (bcd67722a65498b26dd1f0df4b94c301235a3e7b)
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

Stash listemiz

stash pop sonrası listemiz

- **git stash apply** komutu ile istediğiniz değişikliği geri yükleyebilirsiniz. Ancak bu işlem sonrasında yüklediğiniz değişiklik listeden **silinmeyecek**.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$ git stash apply stash@{1}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   dosya1.md
        modified:   dosya2.md
        modified:   dosya3.md

no changes added to commit (use "git add" and/or "git commit -a")
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

apply sonrası stash listemiz

Herhangi bir değişikliği listeden silmek için **git stash drop** komutunu kullanabilirsiniz.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$ git stash drop stash@{1}
Dropped stash@{1} (f887c0e92f469d4791d995b55d998f77c0f9efb8)
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

drop sonrasında stash listemiz

Stash Başka Hangi Durumlarda Kullanılabilir?

Stash işlemini üzerinde çalıştığımız aktif branch'imizi temiz bir duruma getirmek için kullanabiliriz. Bunun dışında aşağıdaki durumlarda da Git'in Stash özelliğini kullanabilirsiniz

- Farklı bir branch'i aktif hale getirmeden önce
- Remote Repository değişikliklerinizi yerel diskinize indirmeden önce
- Branch'inizi merge etmeden önce

Basit Bir Branching Akışı

Gelin şimdi hep birlikte günlük çalışmanız sırasında kullanabileceğiniz basit bir branching akışını ele alalım. Çalışma senaryomuzun şöyle geliştiğini düşünelim

1. Bir web sitesi üzerinde çalışmaya başladınız
2. Bu siteye yeni bir özellik eklemek için bir branch oluşturdunuz
3. Bu yeni branch üzerinden değişikliklerinizi yapmaya başladınız

Bu sırada web sitesinde bir güvenlik açığı tespit edildiğini bildiren bir email aldınız. Acil olarak bu güvenlik açığını gidermeniz için yapmakta olduğunuz çalışmayı bırakmanız ve bu durumu düzeltmeniz gerekiyor. Böyle bir durumda aşağıdaki adımları takip edebilirsiniz

1. Aktif branch'inizi web sitenizin son stabil versiyonunun bulunduğu **master** branch olarak değiştirdiniz.

git checkout master komutunu kullandık

2. Güvenlik açığını giderme çalışmanız için yeni bir branch oluşturdunuz.

- **git branch loginsorunu** komutunu kullanarak branch oluşturduk ve
- **git checkout loginsorunu** komutu ile bu branch'i aktif > hale getirdik

3. Güvenlik açığını giderecek değişikliği tamamladınız, testlerinizi yaptınız ve bu değişikliği Staging Area'ya ekleyip sonrasında da commit ettiniz.

- **git add login.xyz login.html login.css** ile değişiklikleri Staging Area'ya gönderdik
- **git commit -m "Özel karakter içeren kullanıcı adlarında ortaya çıkan güvenlik sorunu giderildi"** ile değişikliklerimizi commit ettik.

4. **master** branch'imizi aktif hale getirdik.

git checkout master komutu ile

5. Commit ettiğiniz değişikliği web sitenizin stabil versiyonunu içeren **master** branch'imize merge ettik.

git merge loginsorunu

6. Daha önce üstünde çalışmakta olduğunuz yeni özellik ile ilgili değişiklikleri içeren branch'inizi aktif hale getirerek çalışmanıza kaldığınız yerden devam edebilirsiniz.

git checkout yeniozellik_xyz komutu ile

Checkout, HEAD ve Working Copy kavramları

Git'de bir branch otomatik olarak o branch için yaptığınız son commit işlemine bir işaretçi tutar ve hangi dosyaların o branch'e ait olduğunu bilir. Herhangi bir anda bir proje için tek bir branch **aktif** olabilir. Bu branch'e **HEAD** denir ve Working Copy içindeki (Working Copy'yi projenizin yerel diskinizdeki dosyalarının tamamı olarak düşünebilirsiniz) dosyalar aktif olan branch'e yani **HEAD**'e aittir. Diğer branch'lerinizdeki dosyalar diskiniz üzerinde değil Git'in veri tabanında (.git klasörü içinde özel bir formatta) bulunur.

Farklı bir branch'i aktif hale getirmek için **git checkout** komutu kullanılır. Bu durumda Git otomatik olarak sizin için iki şey yapar

1. Aktif hale getirdiğiniz branch'i **HEAD** yapar ve
2. Aktif hale getirdiğiniz branch'e ait dosyaları Git veri tabanınızdan yerel diskinize kopyalar ve önceki branch'e ait dosyaları diskinizden kaldırır. Yani Working Copy'nize yeni branch'e ait olan dosyaları koyar.

Değişiklikleri Merge Etmek

Projemizde yaptığımız farklı konular ve bağlamlardaki değişiklikleri takip etmek bir önceki bölümde anlattığımız basit iş akışı ile günlük çalışmamızda bize ciddi kolaylıklar ve esneklikler sunmaktadır. Ancak branch'lerimiz üzerinde değişikliklerimizi tamamlayıp Staging ve Commit işlemlerimizi yaptıktan sonra tüm bu değişiklikleri projemizin stabil versiyonu olan **master** branch ile merge etmemiz gerekiyor (*branch -> [merge] -> master*). Merging en basit anlamda herhangi bir brach'de yaptığımız değişiklikleri **master** branch'imiz ile birleştirme veya **master** branch'e entegre etme işlemidir.

Bir branch'deki değişikliklerinizi sadece **master** branch'iniz ile merge etmek zorunda değilsiniz. Kullandığınız Git çalışma pratiğine bağlı olarak herhangi bir branch'i başka bir branch'e merge edebilirsiniz.

Değişikliklerinizi **master** branch'inize merge etmek durumlardan sadece bir tanesidir, günlük çalışmanız sırasında karşılaşacağınız diğer bir durum ise üzerinde çalıştığınız branch'e **master** branch'deki değişikliklerin merge edilmesidir (*master -> [merge] -> branch*). Bu durumu doğurabilecek aşağıdakilere benzer durumlar ile karşılaşabilirsiniz

- Büyük bir ekipte çalışıyorsunuz ve ekip arkadaşlarınız yaptıkları değişiklikleri sık sık **master** branch'e merge ediyorlar. Bu durumda siz de uzun zamandır üzerinde çalıştığınız branch'in master'dan geri kalmaması için merge işlemi yapmak isteyebilirsiniz.
- Tek başınıza çalışıyorsunuz ancak farklı zamanlarda farklı sebepler ile master branch'e merge ettiğiniz bir çok düzeltme yaptınız. Diğer yandan da daha uzun soluklu bir çalışmanızı ayrı bir branch üzerinde yapıyorsunuz. Üzerinde çalıştığınız branch'in master'daki değişikliklerden geri kalmaması için merge işlemi yapmak isteyebilirsiniz.

Commit'leri değil branch'leri entegre etmek! Git'de değişikliklerinizi merge etme işlemi sırasında kaynak branch'inizde tekil olarak hangi değişiklikleri (commit'ler) merge etmek istediğinizi teker teker söylemezsiniz. Bunun yerine Git'de doğrudan kaynak branch'inizin tamamını hedef branch'e merge edersiniz, çünkü git hangi değişikliklerin hedef branch'de bulunmadığını otomatik olarak tespit edip sadece bunların entegre edilmesini sağlar. Kaynak branch'deki değişiklikler her zaman HEAD'e yani aktif branch'iniz hangisi ise ona entegre edilir.

Git'de merge işlemi çok basit iki adımda yapılır.

1. **git checkout** komutu ile değişikliklerin aktarılacağı hedef branch'inizi aktif (HEAD) hale getirirsiniz.
2. **git merge** komutu ile kaynak branch'deki commit edilmiş değişiklikler HEAD'e entegre

edilir.

```
Alis-MacBook-Pro-2:git101 aliozgur$ git checkout master
Switched to branch 'master'
Alis-MacBook-Pro-2:git101 aliozgur$ git merge loginsorunu
Updating de18052..5d903e9
Fast-forward
 dosya1.md | 3 ++-
 dosya2.md | 4 +++-
 dosya3.md | 3 ++-
 3 files changed, 7 insertions(+), 3 deletions(-)
Alis-MacBook-Pro-2:git101 aliozgur$
```

Merge işleminden sonra **git log** komutunu çalıştırdığınızda ise hangi değişikliklerimizin (commit) **master** branch'imize entegre edildiğini (merge) kolayca görebilirsiniz.

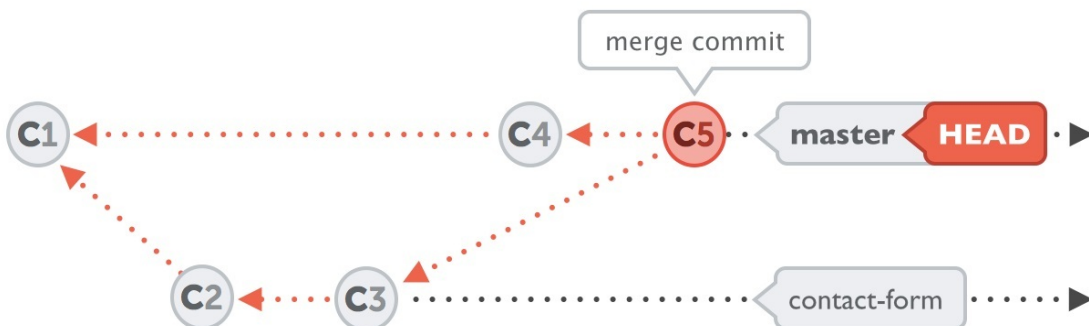
```
Alis-MacBook-Pro-2:git101 aliozgur$ git log
commit 5d903e91bd015d0fda4f4ddd1c6efc413b62cf37
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Tue Aug 26 19:38:47 2014 +0300

    özel karakterli kullanıcı adlarından kaynaklanan sorun giderildi

commit de18052989f96908ef1fcc82b0c507e9a18c499f
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sat Aug 23 21:31:17 2014 +0300

    İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

Ancak Git merge işlemini her zaman bu kadar sade bir şekilde yapamaz, yani Git her zaman kaynak branch'inizdeki commit'lerinizi HEAD'e sırasıyla entegre edemeyebilir. Bu durum genellikle hedef branch'de ve kaynak branch'de birbirinden bağımsız değişikliklerin yapılması durumunda gündeme gelecektir. Bu durumda Git "merge commit" adı verilen; hedef ve kaynak branch'deki en son commit ile gerçekleşen değişiklikleri birleştiren otomatik bir commit adımı ekledikten sonra merge işlemini gerçekleştirir.



Görsel : Tower-Learn Git sayfasından alıntıdır

Bazı durumlarda Git birden fazla otomatik **merge commit** oluşturmak zorunda kalabilir. Bu durumda sizin hangi **merge conflict** noktasını seçip işlemin devam etmesini istediğinizi belirtmeniz gerekecektir (Merge Conflict Resolution)

Branching İş Akışları

Nasıl kullanıldıklarına bağlı olarak branch'leri iki ana grup altında toplayabiliriz.

Bu grupta sadece anlamsal ve kullanım pratikleri ile ilgili bir gruplandırma, sonuç itibarıyla branch kavramı daha önceki bölümlerde anlattığımız kadar basit bir Git aracıdır

Kısa Vadeli / Konu Bazlı Branch'ler

Daha önceki bölümlerde branch kullanımı noktasında elinizi korkak alıştırmamanız ile ilgili tavsiyelerde bulunduk. Örneğin yeni özellikler kodlarken, bug fix yaparken veya deneysel özellikler ile ilgili çalışırken istediğiniz şekilde kolayca ve hızlı bir şekilde üstelik düşük maliyetli branch'ler oluşturabilirsiniz. Bu tür amaçlar için oluşturulan branch'lerin iki ortak özelliği vardır

- Bu branch'ler tek konu veya değişiklik için oluşturulur. Örneğin size bildirilen bir hata için oluşturduğunuz branch üzerinde "GitHub İle Sisteme Giriş" benzeri yeni bir özelliği kodlamayız.
- Bu branch'ler üzerindeki çalışmanız göreceli kısa sürmektedir. Çalışmamız tamamlandığında bu branch'leri **master** veya daha geniş kapsamda tarif edilen bir branch'e merge edip sileriz.

Uzun Soluklu Branch'ler

İkinci türdeki branch'ler ise daha üst seviyede anlam taşırlar ve yeni özellikler, bug fix ve deneysel çalışmalar gibi odaklanmış konular yerine projenizi stabil, test ve development gibi aşamalarını temsil ederler. Bu tür branch'ler projeniz üzerinde geliştirme yaptığınız sürece varlıklarını sürdüreceklerdir. Tipik olarak bu tür branch'ler ile ilgili aşağıdaki kurallar geçerlidir

- Genelde bu tür uzun soluklu branch'ler üzerinde doğrudan değişiklik yapmazsınız. Çalışmalarınızı kısa vadeli branch'ler üzerinde yaparak değişiklikleri bu branch'lere entegre edersiniz.
- Uzun soluklu branch'ler arasında bir hiyerarşi vardır. Genellikle **master** branch projenizin stabil versiyonudur ve hiyerarşik olarak bir altında geliştirmelerinizi entegre ettiğiniz ve daha az stabil olabilen **development** branch'i yer alır.

Uzun soluklu branch'lerin hangi kriterlere göre oluşturulacağı, nasıl yönetileceği ve isimlerinin ne olacağı genellikle çalışan ekibe ve projeye göre değişebilir. Ancak her hâlükârda nasıl bir branching stratejisinin izleneceğine ekip olarak fikir birliği içinde karar verilmelidir.

Basit ve Faydalı Branching Stratejileri

Yukarıda da belirttiğimiz gibi branch'in stratejileri ekibe ve projeye göre değişebilir, ancak aşağıda çoğu ekip tarafından kullanılabilecek basit bir iş akışı kullanabilirsiniz

Sadece bir tane uzun soluklu branch kullanın

Daha önce de belirttiğimiz gibi birden fazla uzun soluklu branch kullanabilirsiniz ancak çoğu zaman bu tip bir yaklaşım karışıklıklara ve fazladan efor sarf etmek gibi zorluklara sebep olabilir. Tek bir uzun soluklu branch kullanmanız durumunda (genelde **master** ismi kullanılır) işiniz önemli miktarda sadeleşip kolaylaşacaktır.

Bu yaklaşım ile çalışmanız durumunda **master** branch'iniz projenizin stabil kodunu barındırmalıdır. Kodunuzun stabil olmasını garantilemek için **master** branch'e entegre edilen (merge) tüm değişikliklerin testler, kod okuma vs gibi kalite kontrol yöntemleri ile denetlenmesi gerekecektir. Bunun bir yansıması olarak değişikliklerin doğrudan **master** branch üzerinde yapılmaması gibi bir zorunluluk da doğacaktır. Eğer *git checkout master* ve sonrasında *git commit* komutlarını çalıştırıyorsanız bilin ki stabilite kuralını ihlal ediyorsunuz.

Konu Bazlı Branchler'i Bolca Kullanabilirsiniz

Projeniz için yeni bir özellik üzerinde çalışmak için, bug fix yapmak için veya deneysel özellikleri ve iyileştirmeleri kodlamak için ayrı birer branch oluşturmaktan ve bu branch'ler üzerinde değişikliklerinizi yapmaktan imtina etmeyin. Bu yaklaşım nerdeyse tüm branching iş akışlarında çok sık kullanılan ve sizin de alışkanlık haline getirmeniz gereken bir yaklaşımdır.

Sadece bir tane uzun soluklu ve stabil branch'iniz (master) olduğu için konu bazlı branch'lerinizin hepsini bu ana branch'i baz alarak oluşturup değişikliklerinizi tamamlayıp kalite kontrol sürecinizi (testler, kod okuma vs) de işlettikten sonra bu değişiklikleri tekrar ana branch'iniz olan master'a entegre etmelisiniz.

Diğer yandan siz kendi konu bazlı branch'inizde değişiklikleri yaparken ekip arkadaşlarınız da arada kendi değişikliklerini master branch'e entegre ediyor olacaklardır. Bu durumda da kendi branch'inizi master branch'deki değişiklikler nedeniyle güncel tutmak için master'daki değişiklikleri de kendi konu bazlı branch'inize sıkça entegre etmelisiniz.

Bu basit akışta unutmamanız gereken tek bir altın kural var; değişikliklerinizi kalite kontrol süreçlerinizi işletmeden ana branch'iniz olan ve her zaman stabil olması gereken master'a entegre etmeyin aksi durumda master branch'inizin stabilitesini bozabilirsiniz.

Remote ve Yerel Branch'lerinizi Senkronize Edin

Git'te remote ve yerel branch'leriniz pratik olarak birbirinden tamamen bağımsızdırlar. Ancak gündelik çalışmanız sırasında kendi bilgisayarınızda oluşturduğunuz branch'lerin uzaktaki sunucudaki eşleniğinin de olmasını sağlamalısınız.

Değişikliklerinizi Sıkça Remote Branch'lere Yükleyin (Push)

Remote branch'ler ile yerel branch'leri sadece yapısal olarak değil yaptığınız değişiklikler anlamında da senkronize etmelisiniz. Bu şekilde ekibinizin geri kalanı da sizin yaptığınız güncel değişikliklerden haberdar olacak ilave olarak yerel branch'lerinizi yedeğini almış olacaksınız.

Diğer Branching Stratejileri

Bu bölümde bahsettiğimiz basit stratejiler ve iş akışları genelde küçük ve çevik (agile) takımlar tarafından kullanıma uygundur. Daha büyük projelerde ve farklı takım kurgularında daha sıkı kurallar ve daha farklı branching yaklaşımlarının kullanımını gerektirebilir.

Gitflow, Forking ve Pull Request adı verilen alternatif iş akışları ile ilgili arama yaparak farklı yaklaşımları kendiniz inceleyebilirsiniz.

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 ağda 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.

Gelin şimdi Local ve Remote repository'leri birbirinden ayıran temel özelliklere göz atalım

Konum

Local repository'ler geliştiricilerin kendi bilgisayarlarında yer alırken Remote repository'ler, çoğunlukla internet olmak üzere, ekipteki herkesin erişebileceği bir sunucuda yer alırlar.

Özellikler

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 yapılabilir. 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 Oluşturma

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

Local/Remote iş akışı

Git'de remote repository işlemleri için az sayıda komut vardır. Günlük çalışmamız sırasında bölümün başında da belirttiğimiz gibi Git işlemlerimizin çoğu local repositorymiz üzerinde gerçekleşir ve internet veya ağ bağlantısına ihtiyaç duymayız. Ancak remote repository komutlarını kullanabilmek için internet veya ağ bağlantısına ihtiyaç vardır.

Bu bölümümüzde Remote Repository'ler ile ilgili aşağıdaki konuları ele alarak ayrıntıları öğreneceğiz

- Remote Bir Repository'ye Nasıl Bağlantı Sağlanır
- Remote Repository'deki Verilerin İncelenmesi
- Remote Değişiklikleri Entegre Etmek
- Local Bir Branch'i Yayınlamak (Publish)
- Branch'leri Silmek

Remote Bir Repository'ye 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 repository için ise böyle bir bilgi tutulmaz. Ancak bölüm girişinde de ele aldığımız gibi Local bir repository'yi baz alarak yeni bir remote repository oluşturabiliriz. Bunun için **git clone** komutunu kullanabiliriz. Örneğin

```
Alis-MacBook-Pro-2:git101 aliozgur$ git remote add git101_ornek https://github.com/aliozgur/git101_ornek.git
Alis-MacBook-Pro-2:git101 aliozgur$ git remote -v
git101_ornek      https://github.com/aliozgur/git101_ornek.git (fetch)
git101_ornek      https://github.com/aliozgur/git101_ornek.git (push)
Alis-MacBook-Pro-2:git101 aliozgur$
```

Yukarıdaki ekran görüntüsünde ilk komutumuz olan **git remote add** ile local repository'miz ile remote repository'miz arasındaki bağlantıyı kuruyoruz. İkinci komutumuz olan **git remote -v** ile de remote repositorymiz ile ilgili bilgileri görebiliriz.

Dikkat ettiyseniz her bir remote repository için biri **fetch** diğeri de **push** işlemleri için kullanılan iki adres bulunur. **fetch** adresini remote repository'den yapılacak olan okuma işlemleri, **push** adresini de remote repository'ye yapılan yazma işlemleri için kullanılır. Genel olarak bu iki adres aynı olmakla birlikte performans ve güvenlik gibi gerekçeler ile iki farklı adres de kullanılabilir.

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 İncelenmesi

git clone komutu remote bir repository'yi yerel diskimize indirdikten sonra **git branch -va** komutunu çalıştırdığımızda aşağıdaki görüntüde yer alan bilgiler listelenecektir.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -va
  loginsorunu          5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
* master              5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
  remotes/origin/HEAD  -> origin/master
  remotes/origin/master 5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

Dikkat edecek olursanız local repository'lerimiz hala yerinde duruyor ancak listemizde ilave olarak **origin/HEAD** ve **origin/master** isimli iki remote kaydı var. Peki daha önceki bölümde **git add git101_ornek** komutu ile oluşturduğumuz remote repository kayıtlarımız neden listlenmiyor? Bunun nedeni önceki bölümde kullandığımız **git add** komutu ile local ve remote repository arasında sadece bir ilişki/bağlantı tanımladık, aslında bu komut sonrasında local ve remote arasında herhangi bir veri transferi gerçekleşmez.

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.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git fetch git101_ornek
From https://github.com/aliozgur/git101_ornek
 * [new branch]      master    -> git101_ornek/master
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

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 işlemten sonra tekrar **git branch -va** komutunu çalıştırdığımızda **gitornek_101/master** isimli remote repositorymizdeki branchlere ilişkin bilgileri de görebiliriz.

```

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git fetch git101_ornek
From https://github.com/aliozgur/git101_ornek
* [new branch]      master      -> git101_ornek/master
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -va
loginsorunu          5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
* master             5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
remotes/git101_ornek/master 5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
remotes/origin/HEAD      -> origin/master
remotes/origin/master    5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
Alis-MacBook-Pro-2:git101_ornek aliozgur$ 

```

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.

```

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git checkout --track git101_ornek/loginsorunu
Branch loginsorunu set up to track remote branch loginsorunu from git101_ornek.
Switched to a new branch 'loginsorunu'
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -va
* loginsorunu          5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
master                 5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
rm                     5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
remotes/git101_ornek/loginsorunu 5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
remotes/git101_ornek/master    5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
remotes/origin/HEAD          -> origin/master
remotes/origin/master        5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
Alis-MacBook-Pro-2:git101_ornek aliozgur$ 

```

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

Tracking Relationship (Takip İlişkisi): Git'de daha önceki bölümlerde de bahsettiğimiz gibi branchler aslında birbirinden tamamen bağımsızdır ve aralarında doğrudan bir ilişki yoktur. Ancak *track* parametresi ile local bir branch'in hangi remote branch'deki değişiklikleri takip edeceğini tanımlayabiliriz. Bu durumda Git iki branch'den herhangi birinde yer alan ancak diğerinde yer almayan commit'leri tespit ederek bizi bilgilendirecektir. Yani

- Local branch'inizde remote branch'e yayınlamadığınız (push) commit'ler varsa bu durumda local branch'inizin remote branch'den önde (ahead) olduğu
- Takım arkadaşlarınız remote branch'e bazı commitleri push ettiğinde ve siz de local branch'inizi güncellemediğiniz durumda local branch'inizi remote branch'in gerisinde (behind) olduğu bilgisi Git tarafından "Tracking Relationship" tanımı sayesinde **git status** komutunun çıktısı olarak gösterilir

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.

```

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git status
On branch loginsorunu
Your branch is up-to-date with 'git101_ornek/loginsorunu'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   dosya1.md
    modified:   dosya3.md

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git commit -m "klonlanmış local branch'de örnek değişiklik"
[loginsorunu 3ed6c20] klonlanmış local branch'de örnek değişiklik
2 files changed, 4 insertions(+), 2 deletions(-)
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

    git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

    git config --global push.default simple

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 472 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To https://aliozgur:hell666@github.com:aliozgur/git101_ornek.git
  5c16035..3ed6c20  loginsorunu -> loginsorunu
Alis-MacBook-Pro-2:git101_ornek aliozgur$
  
```

1) Değişikliklerimizi yaptık

2) Değişiklikleri commit ettik

3) Değişikliklerimizi remote branch'e gönderiyoruz

git push push komutu aslında **git push** formatındadır. Ancak local branch'imizi oluştururken kullandığımız *track* parametresi sayesinde kurulan "Takip ilişkisi" 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) görmemiz ve incelemeniz gerekir.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git fetch origin
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From https://github.com/aliozgur/git101_ornek
* [new branch]      loginsorunu -> origin/loginsorunu
  5c16035..3751637  master       -> origin/master
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git log origin/master
commit 3751637d5f38da51fe7558d4e7b3ea6f16935460
Author: Ali Özgür <aliozgur79@gmail.com>
Date:   Wed Aug 27 23:23:35 2014 +0300

    Update README.md

commit 45b876b4e8a6ebf7c288cc712b7f70886644ab
Author: Ali Özgür <aliozgur79@gmail.com>
Date:   Wed Aug 27 23:23:11 2014 +0300

    Update dosya1.md

commit 5c160359a6d00c77edb458a48c8afa4b7afeff93
Merge: 5d903e9 9dfa88e
Author: Ali Özgür <aliozgur79@gmail.com>
Date:   Wed Aug 27 21:44:33 2014 +0300

    Merge branch 'master' of https://github.com/aliozgur/git101_ornek

commit 9dfa88e0beee8b716cc2a16ec92d017c65959c6c
Author: Ali Özgür <aliozgur79@gmail.com>
```

1. değişiklik

2. değişiklik

Remote branch'deki değişiklikleri indirmek için **git fetch** komutunu kullanıyoruz. Git fetch komutuna geçilen *origin* değeri ise daha önceki bölümlerde gösterdiğimiz *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 branch'deki değişikliklerin bilgilerini indirmek için kullanılan **fetch** (türkçe anlamı [getirmek](#)) ve bu değişiklikleri entegre etmek için kullanılan **pull** (türkçe anlamı [çekmek](#)) ifadelerinin birbirine yakın anlamları olduğu için karıştırabilirsiniz. Bu karışıklığın önüne geçmek için yapacağınız en güzel şey **git pull** komutunu hiç kullanmamak olacaktır. Ayrıntılar için İngilizce bir blog post olan [Git: fetch and merge, don't pull](#) inceleyebilirsiniz.

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**

İlerleyen bölümlerde çakışmaların tespit edilmesi, çözülmesi ve değişikliklerin entegre edilmesi konularını ayrıntılı olarak ele alacağız şimdilik sadece iş akışımızı özetleyip bu konuyu burada sonlandıralım. Akışımız özetle şöyle olacak

- git fetch : remote'dan güncelleme bilgilerini indir
- git diff : remote ve local arasındaki farkları incele
- git merge : değişiklikleri otomatik merge et çakışma varsa bir sonraki adıma geçin
- Çakışma olan dosyalarınızı açın ve çakışmaları düzeltin
- git add: çakışmanın giderildi ve değişiklik Staging Area'ya alındı

Local Bir Branch'i Yayınlamak (Publish)

Kendi bilgisayarınızda oluşturduğunuz Local bir branch siz yayınlamaya karar vermediğiniz sürece sadece sizin bilgisayarınızda yer alacaktır. Yani local bazı branchlerinizi sadece kendi bilgisayarınızda tutarken istediklerinizi de takım arkadaşlarınız ve hatta tüm dünya ile paylaşabilirsiniz.

Gelin şimdi **superyeniozellik** isimli local branch'i remote repositorymizde paylaşalım.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git checkout superyeniozellik
M      dosya1.md
Switched to branch 'superyeniozellik'
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git push -u origin superyeniozellik
Total 0 (delta 0), reused 0 (delta 0)
To https://aliozgur:hell666@github.com/aliozgur/git101_ornek.git
 * [new branch]      superyeniozellik -> superyeniozellik
Branch superyeniozellik set up to track remote branch superyeniozellik from origin.
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

Önce **git checkout** komutu ile branch'imizi aktif hale getiriyoruz ve sonra **git push** komutu ve **-u** seçeneği ile local branch'imizi remote repository'mizde yayınlıyoruz. Push komutu için verdiğimiz **origin** ve **superyeniozellik** değerleri ile **HEAD** branch'imizi **origin** remote repository'de **superyeniozellik** isimli branch olarak yayınlanmasını istediğimizi tanımlıyoruz. **-u** seçeneği ise local branchimiz ile remote branchimiz arasında, önceki bölümlerde de bahsettiğimiz, Takip İlişkisi (Tracking Relationship) kurulmasını sağlar.

git branch komutunu **-vva** seçeneği ile çalıştırdığınızda kurulmuş Takip İlişkisi bilgilerini de görebilirsiniz.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -vva
loginsorunu      8a675af [git101_ornek/loginsorunu: ahead 3] Merge conflict örneği
master           3751637 [origin/master] Update README.md
rm               5c16035 Merge branch 'master' of https://github.com/aliozgur/git101_ornek
* superyeniozellik 3751637 [origin/superyeniozellik] Update README.md
remotes/git101_ornek/loginsorunu 3ed6c20 klonlanmış local branch'de örnek değişiklik
remotes/git101_ornek/master      3751637 Update README.md
remotes/origin/HEAD              -> origin/master
remotes/origin/loginsorunu       3ed6c20 klonlanmış local branch'de örnek değişiklik
remotes/origin/master            3751637 Update README.md
remotes/origin/superyeniozellik  3751637 Update README.md
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

tracking relationship

Local branch'i remote repository'de yayınladıktan sonra local branch'de yaptığımız değişiklikleri **git push** komutunu parametresiz kullanarak remote branch'imizde yayınlatabiliriz.

Artık remote repository'ye erişim yetkisi olan herkes **superyeniozellik** isimli bu branchinizi görebilir ve bu branch'i baz alarak kendi değişiklikleri üzerinde çalışma yapabilir.

Branch'leri Silmek

Bir önceki bölümde oluşturduğumuz **superyeniozellik** isimli branch üzerindeki çalışmamızı tamamlayıp kalite kontrol sürecimizi de işlettikten sonra bu değişiklikleri **master** 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, remote repository'den silmek için de **git branch -dr superyeniozellik** komutunu kullanabiliriz.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -d superyeniozellik
Deleted branch superyeniozellik (was 3751637).
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git branch -dr origin/superyeniozellik
Deleted remote branch origin/superyeniozellik (was 3751637).
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

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.

Remote branch'i **git branch -dr** komutu ile sildiğiniz halde remote repository'ye erişip branchleri kontrol ederseniz **superyeniozellik** isimli branch'in sunucuda hala durduğunu 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 repository'de yayınlamanız gerekiyor.

[Daha ayrıntılı bilgi için bakınız \(StackOverflow - İngilizce \)](#)

İleri Seviye Komutlar ve İşlemler

Bu bölümde aşağıdaki ileri seviye işlemleri ve ilişkili komutları ele alacağız

- Değişikliklerinizi Geri Almak
- Versiyonlar Arasındaki Farkları İncelemek
- Çakışmaları Gidermek
- Merge Alternatifi Olarak Rebase Kullanımı

Değişikliklerinizi Geri Almak

Git'in en güzel yanlarından biri de yaptığınız herhangi bir değişikliği kolayca geri alabilmemizi sağlamasıdır.

Son Commit Bilgilerini Düzeltmek

Commit işlemlerinizi ne kadar dikkatli yaparsanız yapın bazen commit'e dahil etmeyi unuttuğunuz veya yanlışlıkla dahil ettiğiniz dosyalar olabilir veya commit mesajında eksik bilgi vermiş olabilirsiniz. Bu durumda son commit işleminizi yeniden yapmak için **git commit** komutunu **--amend** seçeneği ile kullanabilirsiniz. Sadece commit mesajınızı değiştirmek istiyorsanız **--amend -m** seçenekleri ile git commit komutunu çalıştırabilirsiniz, eğer son commit'e dosya eklemek veya dosya çıkarmak isterseniz commit komutundan önce önceki bölümlerde de bahsettiğimiz **git add** ve **git rm** komutları ile önce Staging işlemini yapabilirsiniz.

Versiyon Kontrolünün Altın Kuralları

#5 Asla Yayınlanmış Commitlerinizi Düzeltip Tekrar Yayınlamayın

****git commit**** komutunun **--amend** seçeneği commit hatalarımızı hızlıca ve kolayca düzeltebilmemiz için oldukça faydalı bir seçenektir. Ancak bu seçeneği kullanmadan önce aşağıdaki noktaları dikkate almalısınız

- Bu seçenek sadece son commit işlemimizi düzeltmemizi sağlar, önceki commitlerimizi bu seçenek ile düzeltemeyiz.
- Bu seçenek ile commit işlemi sonrasında bir önceki commit işlemine dair bilgiler silinir. Proje üzerinde çalışan tek kişi iseniz bu seçeneği kullanmanız sorun yaratmayacaktır ancak bir takım içinde yer alıyorsanız diğer takım arkadaşlarınız sonradan **--amend** ile düzelttiğiniz hatalı commit işleminizi baz alarak kendileri de değişiklikler yapmış olabilirler. Bu durum takım arkadaşlarınız için sorun oluşturacaktır, çünkü onların baz aldıkları commit ile ilgili Git'de artık herhangi bir kayıt yer almayacak.

Local Değişiklikleri Geri Almak

Henüz commit etmediğimiz değişikliklere Local değişiklik denir. Bazen önceki halinden daha kötü olan kod yazabilirsiniz ve bu değişikliği geri almak isteyebilirsiniz. Bu gibi durumlarda değiştirdiğiniz halinden memnun olmadığınız dosyadaki değişiklikleri geri alıp dosyanın son commit edilmiş haline geri dönmek istediğinizde, önceki bölümlerde de sıkca kullandığımız, **git checkout** komutunu `--` seçeneği ile çalıştırmanız yeterli olacaktır.

- `$ git checkout -- dosya1.md` veya
- `$ git checkout -- klasor/dosya2.md` şeklinde kullanabilirsiniz.

Tüm dosyalarda yaptığınız değişiklikleri geri almak istiyorsanız **git reset** komutunu `--hard` seçeneği ile kullanabilirsiniz

`$ git reset --hard HEAD`

Bu komut ile Git tüm dosyaların son commit edilen değişiklikleri içeren HEAD versiyonundaki hallerinin Working Copy'nize geri yükler.

git checkout -- ve **git reset --hard** komutları sonrasında kayıt altına alınmamış olan tüm değişiklikler geri dönüşü olmayacak şekilde yok olur. Bu nedenle bu komutları çalıştırırken dikkatli olmalısınız ve iki defa düşünmelisiniz.

Commit Edilen Bir Değişikliği Geri Almak

Hatalı bir düzenleme yaptığınızda (ki bu genelde test edilmeden yapılan commit'ler sonrasında oluşan bir durumdur) veya geliştirdiğiniz bir özelliğin artık gerekli olmadığına karar verildiğinde yaptığınız değişikliği geri almanız gerekecektir.

git revert komutu commit ettiğiniz herhangi bir değişikliği geri almak için kullanılır. Bu komut ile commit işleminizin kendisi veya bilgileri silinmez sadece commit işleminizdeki değişiklik geri alınır. Örneğin eklediğiniz bir satırı kaldırmak isterseniz **git revert** komutu ile bunu yapabilirsiniz. Aslında git revert komutu değişikliğinizi geri almak için otomatik olarak yeni bir commit oluşturur ve geri alma işlemi bu commit sayesinde değişiklik tarihçesinde görünür hale gelir.

```

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git revert 55b57b
[master 374c6f0] Revert "degisiklige gerek kalmadigi icin geri aliyorum"
1 file changed, 1 insertion(+), 1 deletion(-)
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git log
commit 374c6f0b291e53d3a5d59a2af94e63f741ad677b
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sat Aug 30 21:45:23 2014 +0300

```

```

Revert "degisiklige gerek kalmadigi icin geri aliyorum"

This reverts commit 55b57b8cba87d7d3a924220a32f9f8c072795b6f.

```

```

commit 55b57b8cba87d7d3a924220a32f9f8c072795b6f
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Sat Aug 30 21:44:51 2014 +0300

```

bu değişikliği geri alacağız

```

commit 3751637d5f38da51fe7558d4e7b3ea6f16935460
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Wed Aug 27 23:23:35 2014 +0300

```

Update README.md

```

commit 45b876b4e8a6ebfbc7c288cc712b7f70886644ab
Author: Ali Özgür <aliozgur79@gmail.com>
Date: Wed Aug 27 23:23:11 2014 +0300

```

Yukarıdaki ekran görüntüsünde ilk önce **git revert** komutunu çalıştırdık. Bu komutun en önemli parametresi geri almak istediğimiz commit'in hash değeri (hash'in ilk yedi karakterini kullanabiliriz). Komutu çalıştırdıktan sonra değişiklik tarihçesini incelediğimizde git'in otomatik olarak bir commit oluşturduğunu ve bu commit'in bilgilerinde hangi değişikliğin geri alındığına dair ayrıntıların yer aldığını görüyoruz.

Değişiklikleri geri almak için kullanabileceğimiz diğer bir komut ise **git reset** komutu. Bu komut da herhangi bir bilginizi silmeden işlemi gerçekleştirir, ancak git revert komutundan farklı olarak otomatik yeni bir commit üretmeden değişikliğinizi geri almanızı sağlar.

```

Alis-MacBook-Pro-2:git101_ornek aliozgur$ git reset --hard 374c6f
HEAD is now at 374c6f0 Revert "degisiklige gerek kalmadigi icin geri aliyorum"
Alis-MacBook-Pro-2:git101_ornek aliozgur$ 

```

Bu komut için de git revert komutunda olduğu gibi geri almak istediğimiz commit'in hash değerini veriyoruz. Kullandığımız diğer bir seçenek olan **--hard** seçeneği ise local tüm commitlerinizi silerek geri alma işleminin yapılmasına neden olur, bu nedenle **--hard** seçeneğini kullanırken dikkatli olmalısınız. Local commit'lerinizin korunmasını istiyorsanız **--keep** komutunu kullanabilirsiniz.

30 gün iade garantisi! git reset komutu ile geri alma işlemi sonrasında geri aldığınız noktadan sonraki tüm değişiklikler tarihçeden silinecektir. Ancak git bu silinen bilgileri 30 gün kadar veritabanında tutmaya devam edecektir. Eğer yanlışlıkla geri alma işlemi yaptığınızı farkederseniz 30 gün içinde silinen herhangi bir commit'inizi geri alabilirsiniz.

- [30 günlük süre nasıl değiştirilebilir](#)
- [Silinen commit hangi komutlar ile geri alınır](#)

Versiyonlar Arasındaki Farkları İncelemek

Daha önceki bölümlerde bolca kullandığımız **git status** ve **git log** komutları yaptığınız değişiklikler ile ilgili önemli bilgiler sunar. Ancak bu iki komut ile sadece değişikliklerimizin genel bilgilerini görebiliriz, dosyalarımızda yaptığımız değişikliklerin ayrıntılarını bu komutlar ile göremeyiz. Git'de bu iki komut dışında değişiklikleri ve farkları incelemek için farklı komutlar da yer alır.

İki versiyon arasındaki farkları yorumlamak

Versiyon kontrol sistemlerinde iki versiyon arasındaki değişikliklere İngilizce difference (fark) kelimesinin kısaltması olan **diff** denir. Git'de iki versiyon arasındaki farkları görmek için **git diff** komutunu kullanabilirsiniz. Örneğin **git diff 374c6f..5d903e dosya1.md** komutu ile dosya1.md dosyasının 374c6f ve 5d903e hash'li commitlerdeki iki versiyonunun diff'ini alıyoruz.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git diff 374c6f..5d903e dosya1.md
diff --git a/dosya1.md b/dosya1.md
index eb9cf85..637bdc6 100644
--- a/dosya1.md
+++ b/dosya1.md
@@ -1,4 +1,2 @@
 # Dosya 1
 -başkaca bir değişiklik
 -
-takım arkadaşımız bu satırı değiştirdi
+başka bir değişiklik
 \ No newline at end of file
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

git diff komutunu çalıştırdığımızda yukarıdaki gibi bir ekran ile karşılaşacaksınız. Gelin şimdi bu ekranda numaralandırdığımız önemli alanlarda hangi bilgilerin bize gösterildiğini ele alalım

1. **Karşılaştırılan Dosyalar (A/B):** Diff komutu iki dosyayı birbiri ile karşılaştırır, A dosyası ve B dosyası. Bu A ve B dosyaları genelde aynı dosyanın (bizim örneğimizde dosya1.md) farklı versiyonlarıdır. Çok sık olmasa da diff işlemi ile tamamen farklı olan dosyaları (örneğin dosya1.md ve dosya1_enyeni.md) da karşılaştırabilirsiniz. Hangi dosyaların karşılaştırıldığını açıkça belirtmek için diff komutunun çıktısı her zaman hangi dosyanın A hangi dosyanın da B olduğunu belirterek başlar.

Bu bilginin hemen altında **index** ile başlayan satırda pratik olarak pek işinize yaramayacak dosya bilgileri yer alır. Bu bilgilerden ilk ikisi karşılaştırılan versiyonların hash değeri sonuncusu da (1000644) dosya modusudur.

2. **A/B Dosya Simgeleri:** Dosya içeriğinin hangi kısmının A hangi kısmının da B

dosyasına ait olduğunu belirtmek için kullanılan - ve + sembollerinden hangisinin hangi dosyaya ait olduğu bilgisi.

3. **Fark İşaretçileri:** Diff komutu ile sadece iki dosya (aslında versiyon da denilebilir) arasındaki farkların olduğu satırlar gösterilir, dosyanın tamamı (değişmeyen satırlar da dahil) gösterilmez. @@ simgeleri ile başlayan satırda A ve B dosyaları arasındaki farklı satırların hangi satırdan başlayıp kaç satır olduğu bilgisi gösterilir. Bizim ekran görüntümüzde yer alan @@ -1,4 +1,2 @@ bilgisi bize şunu söyler

(-) simgesi ile tanımlanan A dosyasından 1. satırdan başlayarak 4 satır, + simgesi ile tanımlanan B dosyasından 1. satırdan başlayarak 2 satır birbirinden farklı

4. **Değişiklikleri Okumak :** Değişen her satırın başında (-) veya (+) simgesi yer alır. Bu simgeler ile A ve B versiyonlarının içeriğinin ne olduğunu anlamamızda bize yardımcı olacaktır. Örnek ekran görüntüsünde (-) ile başlayan ve A versiyondaki satırların daha sonra (+) ile başlayan B versiyonundaki satırlar ile değiştirildiğini görüyoruz.

Local Branch'deki farkları incelemek

Daha önceki bölümlerde **git status** komutu ile Local branch'imizde hangi dosyaların değiştiğini görebileceğimizi öğrenmiştik. **git status** komutu ile dosyaların içeriğindeki değişiklikleri göremeyiz. İçerik değişikliklerini de görmek için doğrudan **git diff** komutunu herhangi bir parametre veya seçenek belirtmeden kullanabilirsiniz.

```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git diff
diff --git a/dosya1.md b/dosya1.md
index eb9cf85..2623103 100644
--- a/dosya1.md
+++ b/dosya1.md
@@ -1,4 +1,3 @@
 # Dosya 1
-başkaca bir değişiklik

-takım arkadaşımız bu satırı değiştirdi
+Yeniden oluşturduğumuz içerik.
\ No newline at end of file
diff --git a/dosya2.md b/dosya2.md
index c397d44..3a58e9e 100644
--- a/dosya2.md
+++ b/dosya2.md
@@ -1,3 +1,2 @@
 # Dosya 2
-Birkaç değişiklik yapalım
-login sorunu düzenlemesinin ilk kısmı bu dosyada....
\ No newline at end of file
+Tamamen yeni bir içerik.
\ No newline at end of file
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```

Sadece Staging Area'ya commit edilmek üzere eklenmiş/çıkarılmış dosyalardaki değişiklikleri görmek isterseniz **git diff --staged** komutunu kullanabilirsiniz.

Commit edilmiş dosyalardaki farkları görmek

git log komutunu commit işlemleri ile ilgili özet bilgileri görmek için kullanabiliriz. Bu komutu herhangi bir parametre veya seçenek belirtmeden kullanırsanız dosya içeriğindeki farkları göremezsiniz. Dosyaların içeriğindeki farkları da görmek için *git log* komutunu **-p** seçeneği ile kullanabilirsiniz.

```
$ git log -p şeklinde
```

İki Farklı Branch'i Karşılaştırmak

İki farklı branch'in arasındaki içerik farklarını görmek için *git diff* komutuna karşılaştırmak istediğiniz branch isimlerini parametre olarak verebilirsiniz. Örneğin **master** ile **superyeniozellik** branch'ini karşılaştırmak için *git diff* komutu aşağıdaki gibi olacaktır

```
$ git diff master..superyeniozellik
```

Branch'leri karşılaştırabildiğiniz gibi iki farklı versiyon arasındaki tüm dosyaların içeriğini de versiyonların **hash** değerlerini *git diff* komutuna parametre olarak vererek karşılaştırebiliriz. Örneğin

```
$ git diff 74c6f..5d903e
```

komutu ile 74c6f hash değeri olan commit (versiyon) ile 5d903e hash değerine sahip commit'in dosyaları arasındaki farkları görebilirsiniz.

Çakışmaları Gidermek

Versiyon kontrolü ile ilgili insanların en sevmedikleri ve korktukları şey değişiklikleri entegre etme (merge) işlemi sırasında oluşan çakışmalar ve bu çakışmaların çözülmesi sürecidir. Bu bölümde çakışmalardan korkmamamız gerektiğini ve çakışmaları en kolay ve efektif bir şekilde nasıl çözebileceğimizi ele alacağız.

Git ile güvendesiniz

Git ile çalışıyorsanız istediğiniz zaman yanlış yaptığınız değişiklik entegre etme işlemi geri alarak bu işleme temiz dosyalar ile yeniden başlayabilirsiniz. Bu konuda Git'e güvenmeniz yeterli olacaktır. Merge işlemi sırasında işin çoğunu Git sizin yerinize otomatik olarak yapacak ve size sadece basit çakışmaları çözmek kalacaktır. Git'in diğer bir güzel tarafı ise çakışmaların sadece kendi local branch'inizde olması ve hiçbir zaman sunucu tarafında olamamasıdır. Böylece merge işlemi sırasında meydana gelen çakışmada takım arkadaşlarınız etkilenmeyecektir.

Çakışma Nasıl Oluşur?

Git'de **merge** işlemi başka bir branch'deki değişiklikleri üzerinde çalıştığınız kendi branch'inize entegre etme işlemidir. Git merge işlemi sırasında değişikliklerin çoğunu sizin için otomatik olarak entegre eder.

Ancak bazı durumlarda Git merge işlemi otomatik olarak gerçekleştiremez ve sizin müdahale ederek hangi değişikliğin nasıl entegre edileceğine karar vermeniz gerekir. Bu durum genellikle aynı dosya üzerinde değişiklikler yapıldığında ortaya çıkar, bu durumda bile Git dosyadaki değişiklikleri nasıl entegre edileceğine çoğu zaman otomatik karar verebilir. Fakat aynı satırda yapılan değişiklikler veya takımdaki bir kişinin bir satırı silmesi durumunda sizin bu değişikliği kendi branch'inize nasıl entegre edileceğine karar vermeniz gerekir. Bu durumda Git dosyanızı conflicted (çakışmalı) olarak işaretler ve sizin çalışmanıza devam edebilmeniz için bu çakışmayı çözmeniz gerekir.

Çakışmaları Nasıl Çözeriz

Çakışma oluştuğunda ilk yapmanız gereken şey çakışmanın neden olduğunu anlamak olmalıdır. Örneğin takım arkadaşınız aynı dosyada sizin de değiştirdiğiniz bir satırı mı değiştirdi veya aynı dosyada bir satır mı sildi veya sizinle aynı isimli yeni bir dosya mı oluşturdu?

git status komutunu çalıştırdığınızda Git size branch'inizde entegre edilmemiş dosyalar olduğunu söyleyecektir.


```
Alis-MacBook-Pro-2:git101_ornek aliozgur$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 3 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)

You have unmerged paths.
(fix conflicts and run "git commit")

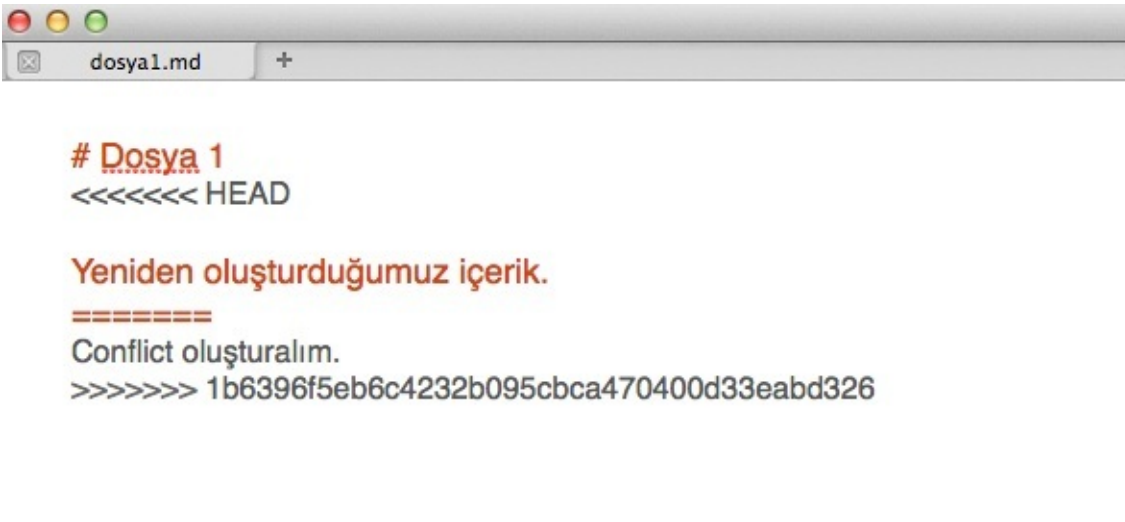
Unmerged paths:
(use "git add <file>..." to mark resolution)

    both modified:    dosya1.md

no changes added to commit (use "git add" and/or "git commit -a")
Alis-MacBook-Pro-2:git101_ornek aliozgur$
```



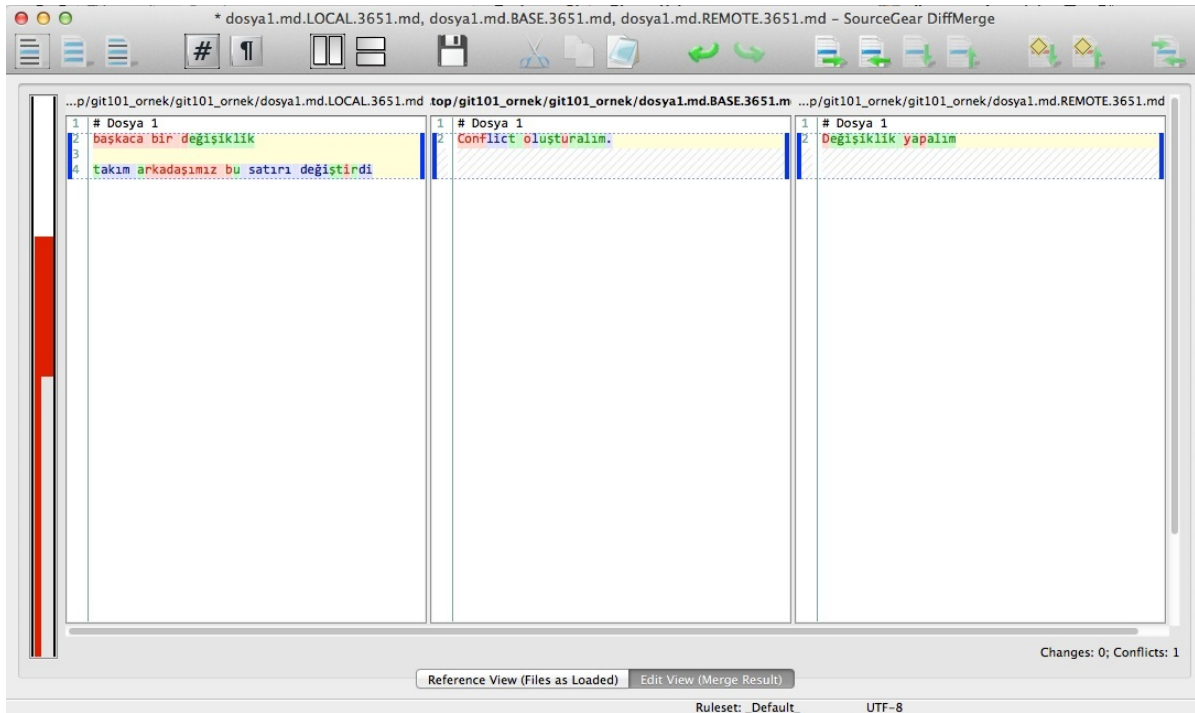
Yukarıdaki ekran görüntüsünde **dosya1.md** isimli dosyamızda çakışma olduğunu görebiliriz. Bu çakışmayı düzeltmek için dosyamızı açıp çakışan satırları düzeltmemiz gerekiyor.



dosya1.md dosyasını açtığımızda yukarıdakine benzer bir görüntü ile karşılaşyoruz.

- <<<<<<< HEAD ile başlayan ve ===== kadar devam eden kısım dosyanın bizim branch'imizde olan versiyonuna ait
- ===== belirtecinden sonraki kısım da değişiklikleri entegre etmek istediğiniz branch'de yer alan dosyanın içeriğini gösterir.

\$ **git mergetool dosya1.md** komutunu çalıştırarak önceki bölümlerde konfigürasyon ayarlarını yaptığımız DiffMerge uygulamasını da açabilirsiniz.



Dosyamızın içeriğinin ne olacağına karar verip kaydettikten sonra normal bir commit işlemi ile çakışmayı çözme işlemini tamamlıyoruz.

- \$ **git add dosya1.md** ile dosyamızı Staging Area'ya ekliyoruz
- \$ **git commit -m "değişiklikler entegre edildi"** komutu ile de commit işlemini tamamlarız.

Merge İşlemini Nasıl Geri Alabiliriz?

Dosyanızın merge işlemine başlamadan önceki haline istediğiniz zaman geri dönebilirsiniz. Bunun için yapmanız gereken tek şey **git merge --abort** komutunu çalıştırmak.

Merge Alternatifi Olarak Rebase Kullanımı

Merge komutu iki branch arasındaki değişiklikleri entegre etmenin en kolay yolu olmakla birlikte tek yol değildir. Rebase komutu da iki branch'ı entegre etmek için kullanılan merge komutuna alternatif bir komuttur. Bu durumda kafanızda "Neden **merge** yerine **rebase** kullanmak isteyelim?" şeklinde bir soru oluşabilir. Bu sorunun cevabını bulmak için önce gelin **merge** komutunun biraz daha iyi anlamaya çalışalım.

Merge komutuna daha yakın bir bakış

Git merge işlemini gerçekleştirmeden önce aşağıdaki üç commit'i tespit eder

- **İki branch'in ortak commit'i:** İki branch'in de tarihçesini daha yakından incelediğinizde bu branch'lerin zamanın bir noktasında ortak bir commit'e sahip olduklarını görürüz. Bu anda her iki branch'in de içeriği bire bir aynıdır.
- **Branch'lerin son commit'leri:** Her iki branch için de yapılan son commit'ler

Bu üç commit tespit edildikten sonra Git bu üç commit'i birleştirerek entegrasyonu yapabilir.

Fast-Forward ve Merge Commit

Basit bazı durumlarda branch'lerden bir tanesinde herhangi bir değişiklik yapılmamıştır ve bu branch'in yukarıdaki bölümde belirttiğimiz ortak commit'i ve son commit'i aynıdır. Bu durumda merge işlemi çok basitleşir ve git diğer branch'in tüm commit'lerini ortak commit'in üzerine ekleyerek merge işlemini yapar. Bu özel duruma Git terminolojisinde "**Fast-Forward Merge**" denir ve her iki branch'in tarihçesi de ortaktır.

Fakat çoğu zaman her iki branch'de birbirinden bağımsız olarak değişikliğe uğrar ve tarihçe açısından birbirinden uzaklaşırlar. Bu durumda merge işlemini yapmak için Git'in her iki branch arasındaki değişiklikleri içeren otomatik bir commit oluşturması gerekir. Oluşturulan bu commit'e Git terminolojisinde "**Merge Commit**" denir.

Normal Commitleri ve Merge Commitleri Ayırt Etmek

Normal commit'leri yazılım geliştiriciler ince eleyp sık dokuyarak oluştururlar, diğer yandan Merge Commitler ise Git tarafından otomatik oluştururlar. Merge işlemi ile ilgili ayrıntıları daha sonradan incelemek isterseniz her iki branch'in commit tarihçesine ve commit çizelgesine bakmanız gerekir.

Rebase ile değişiklikleri entegre etmek

Bazı takımlar iki branch'i yukarıda anlattığımız otomatik merge commit'ler yerine **rebase** ile entegre etmeyi tercih edebilir. Rebase sonrasında projenizin iki farklı branch'i olduğuna dair herhangi bir tarihsel iz oluşmaz.

Gelin şimdi rebase işleminin nasıl yapıldığına bakalım. Örnek senaryomuzda Branch-B'deki değişiklikleri Branch-A'ya entegre edeceğiz. Rebase işlemini **git rebase** komutunu aşağıdaki gibi kullanarak yapıyoruz.

```
$ git rebase Branch-B
```

Bu komut ile Git öncelikle Branch-A ile Branch-B'nin ortak en son commit'ini bulup ortak commit sonrasında Branch-A'da yapılan diğer tüm commit'leri geri alır. Aslında bu commitler silinmez sadece geçici olarak farklı bir yerde saklanır. Daha sonra Branch-B'deki tüm commitler Branch-A'ya uygulanır. Son aşamada ise Branch-A'nın geçici olarak farklı bir yerde saklanan commit'leri tekrar uygulanır. Bu işlemler sonrasında tüm değişiklikler sanki sadece Branch-A üzerinde gerçekleşmiş gibi görünür.

Git Araç ve Servisleri

Bu bölümde aşağıdaki konulardan bahsedeceğiz

- Görsel Git istemcileri
- Git ile kullanılacak diff/merge araçları
- Git servisleri
- Kaynaklar ve Referanslar

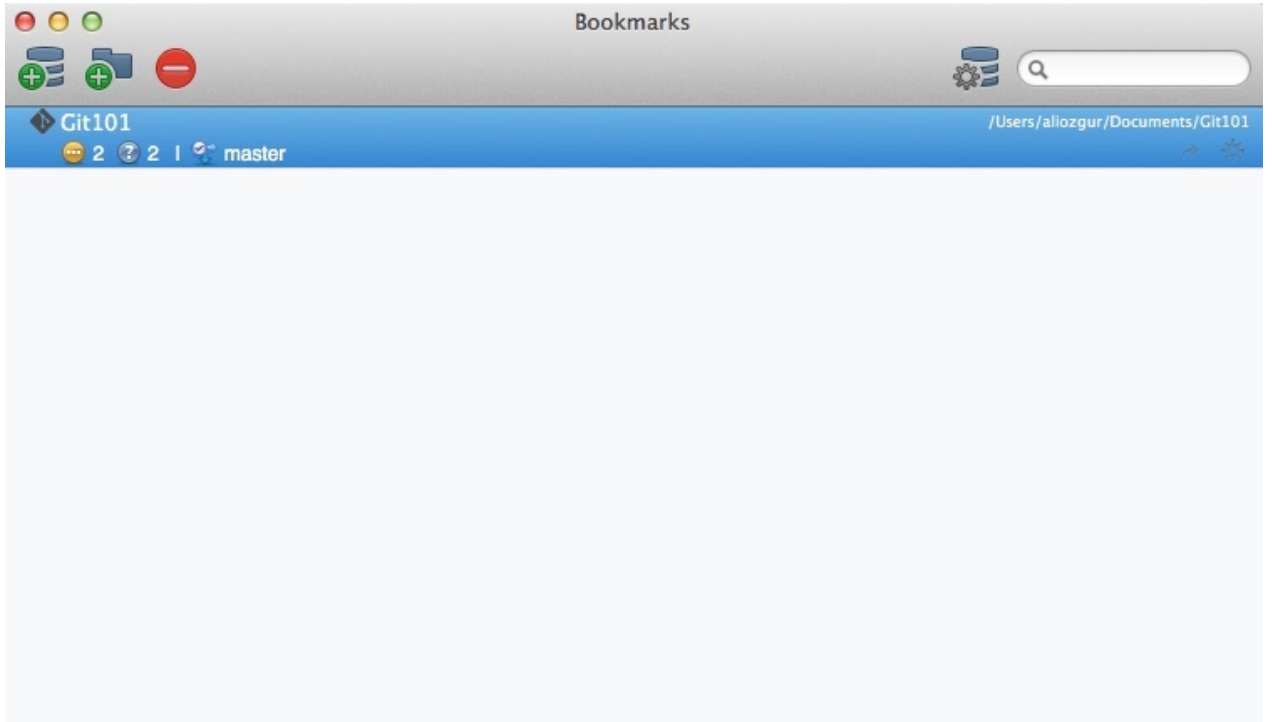
Görsel Git İstemcileri

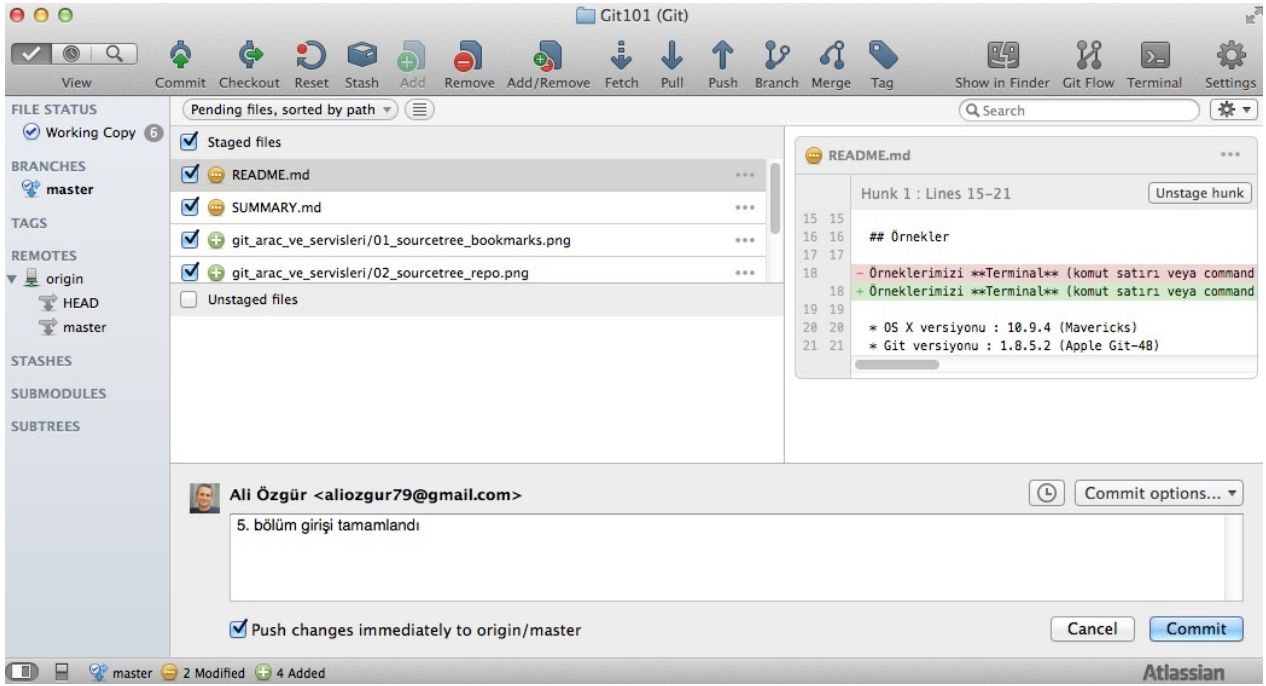
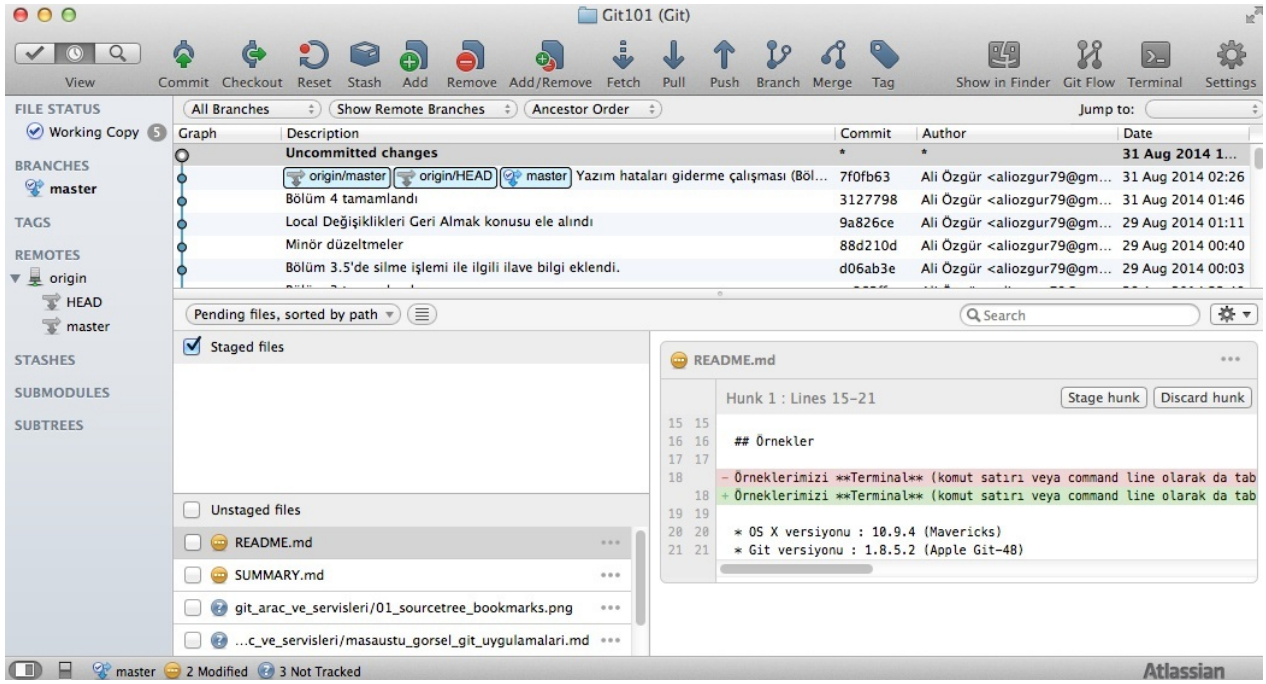
Önceki bölümlerde Terminal kullanarak birçok Git komutunun nasıl kullanıldığını size gösterdik. Ancak günlük çalışmanızda her bir komutun ayrıntılı olarak ne işe yaradığını, hangi parametreler ve seçenekleri kabul ettiğini aklınızda tutmak zor olacaktır. Bu nedenle Git'i günlük iş akışınıza entegre edip Git kavramlarını öğrendikten sonra görsel bir Git istemcisi kullanmak işinizi ciddi oranda kolaylaştıracaktır.

Atlassian SourceTree

SourceTree ücretsiz bir uygulama. SourceTree'yi Mac OS X ve Windows işletim sistemlerinde kullanabilirsiniz.

[SourceTree'yi bu linkten indirebilirsiniz](#)





Tower

Tower, sadece Mac OS X'de çalışan ücretli bir uygulama. [Bu linki](#) kullanarak uygulamanın 30 günlük deneme sürümünü indirebilirsiniz.

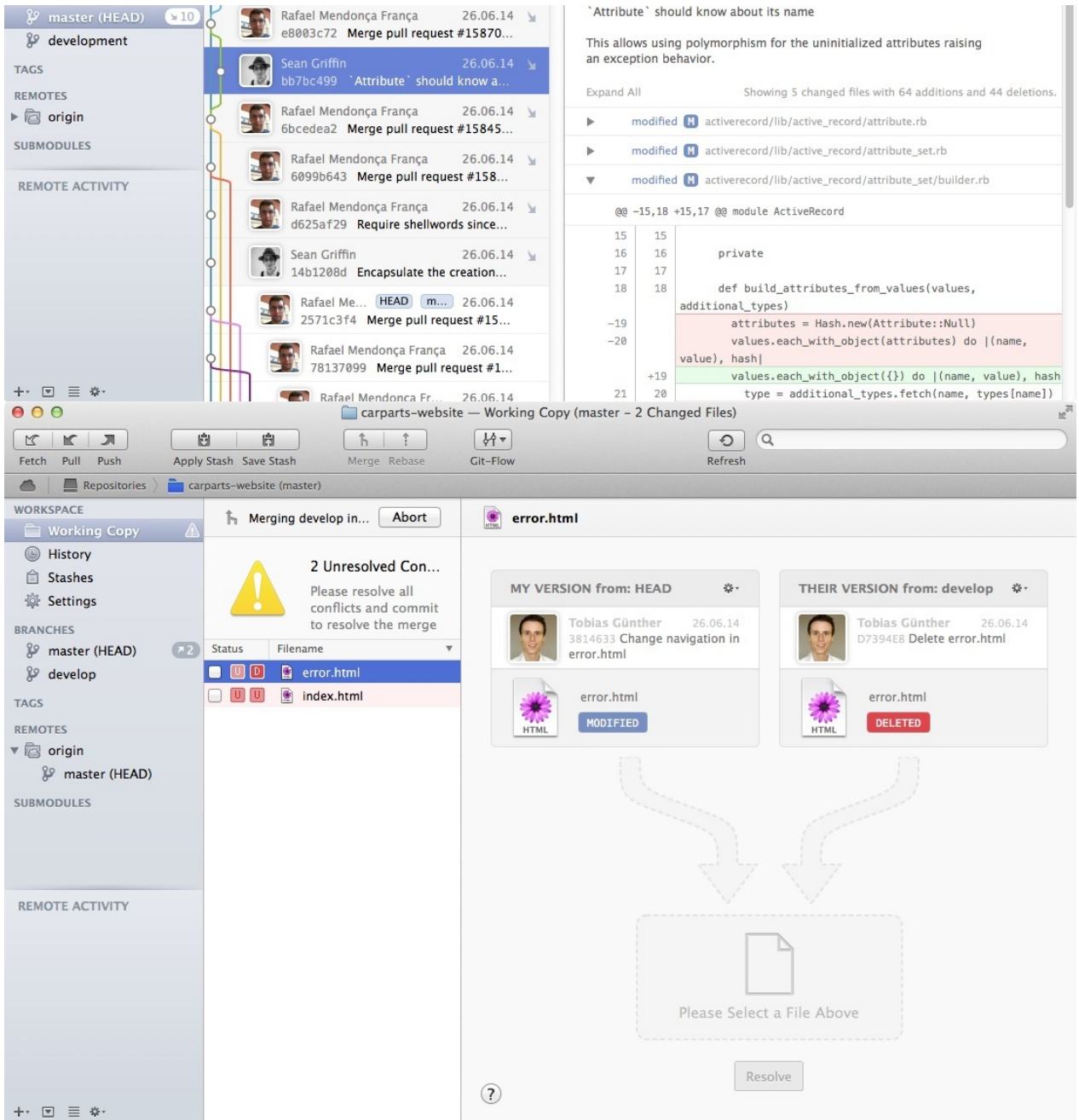
The screenshot displays the Visual Studio Code interface with the **Repositories** view active. The top toolbar includes buttons for **Fetch**, **Pull**, **Push**, **Apply Stash**, **Save Stash**, **Merge**, **Rebase**, **Git-Flow**, and **Refresh**. A search bar for repositories is also present.

The **REPOSITORIES** sidebar on the left shows a tree structure of repositories:

- Concept
- Open Source
 - jQuery
 - Ruby on Rails
- Web Development
 - CarParts Website
 - Tower Website
 - Tower Help
 - Learn Git
 - Blog
- iPhone Development

The main workspace area shows the **carparts-website** repository in the **Working Copy (master - 5 Changed Files)** state. The **index.html** file is selected, showing a diff view with 2 chunks, 3 insertions, and 2 deletions. The diff view highlights changes in the `` tags, showing the addition of a new link and the removal of an old one.

The bottom section of the interface shows the **rails** repository in the **master, origin/master (45011 Commits)** state. The **Changeset** view is active, displaying a commit by **Seán Griffin** with the message **Deprecate automatic count...** and a date of **26.06.14**. The commit hash is **bb7bc499**.



GitHub

Projelerinizin kaynak kodunu [GitHub'da](#) tutuyorsanız GitHub'ın ücretsiz Mac OS X ve Windows için geliştirdiği kullanımı oldukça kolay olan ve Git'in karmaşasından sizi bir nebze olsun uzaklaştırabilecek uygulamasını kullanabilirsiniz. Bu uygulamanın Mac OS X versiyonunu [buradan](#) ve Windows versiyonunu da [buradan](#) indirebilirsiniz.

Windows versiyonunda GitShell isimli Terminal benzeri uygulamanın kurulumu da yer alıyor. Bu nedenle daha karmaşık Git komutları için Terminal benzeri bir deneyim istiyorsanız GitShell'i rahatlıkla kullanabilirsiniz.

TortoiseGit

Windows kullananlar ücretsiz bir uygulama olan Tortoise Git uygulamasını [bu linkten](#) indirip kullanabilirler. Özellikle daha önce Subversion ve TortoiseSVN kullananlar için TortoiseGit benzer bir deneyim sunmaktadır.

Multiplatform İstemciler

- [GitEye](#)
- [SmartGit](#)

Linux'a Özel İstemciler

- [gitg](#)
- [giggle](#)
- [GitForce](#)
- [RabbitVCS](#)

Diğer Uygulamalar

Diğer görsel Git uygulamalarını Git'in [kendi sayfasından](#) inceleyebilirsiniz.

Diff/Merge Araçları

Projenizde neler olup bittiğini anlamak için zaman zaman (aslında bir takım içinde yer alıyorsanız sık sık da denilebilir) dosyaların versiyonları arasındaki farkların ne olduğuna bakmanız ve çakışma durumunda da çakışmaları inceleyip çakışma durumunu gidermeniz gerekir.

4.Bölümde Terminal'den herhangi bir yardımcı uygulamaya gerek kalmadan da Git'in bize bu farkları gösterebildiğine ve bu farklar'ı nasıl okuyabileceğimize değinmiştik. Ancak büyük projelerde Git'in sunduğu bu işlev ile farkları okumak çok kolay olmayacaktır. Bu nedenle farkları daha rahat inceleyebilmek için bu farkları renkler ve formatlama yöntemleri ile görselleştiren araçlardan faydalanmak işinizi kolaylaştıracaktır. Farkları görselleştiren bu uygulamaların nerdeyse tamamı aynı zamanda çakışmaları da görselleştirip merge işlemini de kolayca yapmanız için araçlar sunar.

Windows üzerinde çalışıyorsanız [WinMerge \(ücretsiz\)](#) veya [Araxis Merge \(ücretli\)](#) kullanabilirsiniz.

Mac OS X üzerinde çalışıyorsanız [SourceGear DiffMerge \(ücretsiz\)](#) veya Apple XCode ile birlikte ücretsiz gelen Apple'in FileMerge aracını kullanabilirsiniz.

Git Servisleri

Takım çalışması söz konusu olduğunda en önemli konulardan birisi de kaynak kodunun veya daha genel anlamda dosyaların nasıl paylaşılacağına karar vermektir. Bu noktada iki seçeneğiniz var 1) dosyalarınızı kendi sunucularınız üzerinden paylaşmak veya 2) işi paylaşım ve barındırma hizmeti vermek olan online servisler kullanmak

Dosyalarınızı kendi sunucularınız üzerinden paylaşmanın aşağıdaki gibi avantajları vardır

- Düşük maliyet
- Dosyalarınız kendi sunucularınızdadır
- Git'in veya hangi versiyon kontrol sistemini kullanıyorsanız bu sistemin tüm özelliklerini istediğiniz gibi kullanabilirsiniz

Ancak bu seçeneğin aşağıdaki dezavantajlarını da göz ardı edemeyiz

- Sunucuların çalışır halde ve erişilebilir olmasını sağlamak sizin sorumluluğunuzdadır
- Yedekleme sorumluluğu sizde olacak
- Güvenlik ve yazılım güncellemelerini de sizin takip etmeniz gerekir

Eğer sunucu kaynakları yeterli olan, yedekleme, güncelleme gibi sunucu yönetimi konularında ayrı ve uzman ekibi olan bir kurumda çalışıyorsanız dosyalarınızı kendi sunucularınızda barındırmak ilk tercihiniz olacaktır. Ancak küçük bir girişimseniz veya açık kaynak bir projeniz varsa sunucu yönetimi ile ilgili yeterince uzmanlığınız ve kaynağınız olmayabilir. Bu durumda dosyalarınızı online bir servis üzerinde barındırmak ve buradan paylaşma açmak sizin için daha mantıklı olacaktır.

GitHub

Özellikle açık kaynak projeler için oldukça popüler bir servis olan GitHub'ı kullanabilirsiniz. GitHub açık kaynak projeler için ücretsiz olmakla birlikte, kurumlar ve özel projeler için de oldukça makul fiyatlara Git sunucu hizmet'i sunmaktadır.

[GitHub Ana Sayfa](#)

BitBucket

Daha önce küçük bir girişim olarak Mercurial (bu da dağıtık bir versiyon kontrol sistemi) hizmeti sunmak için kurulan BitBucket Atlassian tarafından satın alındıktan sonra Git sunucu hizmeti de sunmaya başladı. BitBucket açık kaynak veya özel 5 kullanıcıya kadar olan

sınırsız sayıda projeniz için ücretsiz hizmet sunar aynı zamanda oldukça makul fiyatlara da daha fazla kullanıcı için ücretli hizmet seçeneği de var.

[BitBucket Ana Sayfa](#)

Visual Studio Team Services

Microsoft tarafından sunulan bu servis ile 5 kullanıcıya kadar sınırsız sayıda Git Repository'si oluşturabilirsiniz. Daha büyük takımlarınız için yapacağınız aylık ödemelerle de bu servisten yararlanabilirsiniz. Ayrıca Visual Studio Team Services'de, Git'in yanı sıra Microsoft'un kendi versiyon kontrol sistemi olan Team Foundation Server (TFS) ile de kodlarınızı versiyonlayabilirsiniz.

[Visual Studio Team Services Ana Sayfa](#)

Kaynakça ve Referanslar

Git oldukça çok seçeneği ve farklı kullanım şekli olan bir dağıtık bir versiyon kontrol sistemidir. Git ile ilgili daha fazla bilgi edinmek istiyorsanız önce iyi derecede İngilizce öğrenmenizi tavsiye ediyorum. Daha sonra da aşağıdaki kaynaklardan başlayarak Git ile ilgili bilginizi arttırabilirsiniz.

1. [Git-Scm Referans Dokümanı](#)
2. [GitRef Referans Dokümanı](#)
3. [Learn Version Control with Git](#)
4. [Atlassian Git Tutorials](#)
5. [Pro Git Book](#)
6. [Atlassian Git Workflows](#)
7. [Learn Git Branching Online Tutorial Application](#)
8. [Git: fetch and merge, don't pull](#)
9. [Resolving a merge conflict from command line](#)
10. [Adding And Removing Remote Branches – Git Branch](#)

Git ile Versiyon Kontrolü - Günlük Alıştırmalar ile Tekrar

Bu depoda "Git ile Versiyon Kontrolü" kitabında ele alınan konuları da kapsayacak şekilde Git ile versiyon kontrolünü ve Git iş akışlarını kısa pratikler ile günlük çalışma akışımıza dahil etmek için uygulamalı alıştırmalara yer alıyor. 11 gün boyunca her gün sadece 1 saat ayırıp bir uygulama yaparak Git ile ilgili kitapta anlattığımız konulardaki bilginizi pekiştirebilirsiniz.

NOT: İnternet tarayıcınızı kullanarak Git komutlarını çalıştırmak için Code School tarafından hazırlanan [Try Git](#) sayfasını da kullanabilirsiniz

Gün gün alıştırmalar

- [Gün 1 - İstemci Kurulumları](#)
- [Gün 2 - Yerel Depo Oluşturma](#)
- [Gün 3 - Klonlama](#)
- [Gün 4 - Değişiklikleri kaydetme](#)
- [Gün 5 - Değişiklikleri Versiyon Kontrolüne Alma](#)
- [Gün 6 - Commit Edilmiş Değişiklikleri İptal Etmek](#)
- [Gün 7 - Commit Edilmiş Değişiklikleri Silmek](#)
- [Gün 8 - Dal Oluşturmak](#)
- [Gün 9 - Değişiklikleri Birleştirme \(Merge\)](#)
- [Gün 10 - Rebase](#)
- [Gün 11 - Birlikte Çalışma \(Collaboration\)](#)

1.Gün: İstemci Kurulumları

- Putty (Bu kurulum opsiyonel), Putty Git tarafından depolara SSH ile erişmek için kullanılır. 2. adımda kuracağınız Git for Windows SSH için OpenSSH mı yoksa Putty/Plink mi kurmak istediğinizi soracak
- Git for Windows, Git'in Windows'da çalışması için kullanılan paket.
- Görsel istemciler. Aşağıdaki iki istemciden en az bir tanesini kurabilirsiniz.
 - Atlassian SourceTree (Önerilen)
 - CollabNet GitEye

Alıştırma-1: Git kurulumunun doğru yapıldığının test edilmesi

Windows > Start > Git Bash isimli uygulama gelmeli

Git Bash'i çalıştırıp aşağıdaki komutu yazalım

```
git --version
```

Bu komut çalıştığında aşağıdakine bezer bir versiyon bilgisi görmelisiniz.

```
git version 2.9.2.windows.1
```

Alıştırmaları OS X veya Linux işletim sistemlerinde yapıyorsanız komut satırında git --version komutunu çalıştırabilirsiniz.

Alıştırma-2: Git konfigürasyonunun yapılması

Bu alıştırmaı yapmadan önce

- Kitabın 1. bölümündeki Git Konfigürasyonu başlığına tekrara göz atabilirsiniz
- Git servislerinden Github, Gitlab veya Bitbucket üzerinde bir hesap oluşturmalsınız

Git'in global konfigürasyonuna kullanıcı adınızı ve e-posta adresinizi git config komutu ile tanımlayın.

```
git config --global user.name ali.ozgur  
git config --global user.email ali.ozgur@xyz.com.tr  
git config --global core.eol native  
git config --global core.autocrlf true
```

İPUCU: core.eol ve core.autocrlf değerleri Windows platformu için önerilen değerler. Git'in windows'da End-Of-Line karakterlerini (Windows EOL için CRLF kullanıyor, Linux, Unix ve OSX ise LF kullanıyor) düzgün çalışması için global git konfigürasyonunuzda core.eol ve core.autocrlf ayarlarının yapılması gerekiyor. Ancak, proje depolarınızda .gitattributes dosyasının ilk satırında text=auto değerini girerseniz bu ayar yukarıda yapılan core.eol ve core.autocrlf ayarlarını ezecektir.

Alıştırma-3: Uzak deponun bilgisayarınıza klonlanması

Önceki alıştırmada oluşturduğunuz Github, Gitlab veya Bitbucket hesabınız için ilgili servis sağlayıcısının sayfalarını kullanarak uzak sunucuda boş bir proje deposu oluşturun. Örneğin, Git101 isimli boş bir depo oluşturabilirsiniz. Daha sonra da SourceTree veya GitEye uzak depoyu bilgisayarınıza kopyalayın.

Kopyalama işlemini alternatif olarak git clone komutu ile komut satırı ara yüzünden (Git Bash veya Terminal) de yapabilirsiniz. Clone komutu kullanmak isteyenler

- Kullanıcı adı ve şifre bilgisinin bu komutta nasıl geçileceğini
- Klonlama sırasında karşılaşılabileceğiniz fatal: index-pack failed şeklindeki hatayı gidermek için ne yapılması gerektiğini araştırabilir

[İleri >>](#)

2.Gün: Yerel Depo Oluşturma

Git ile çalışabilmek için illa bir sunucu kurulumuna veya bir Git servis sağlayıcısına ihtiyaç duyulmaz. Git kurulumumuzu tamamladığımıza göre artık kendi bilgisayarımızda da bir depo oluşturup denemelerimizi yapabiliriz.

Bilgi Kutusu (NOT) : Git'in push, pull gibi birkaç komutu dışındaki tüm komutlar kendi bilgisayarınızda çalıştırdığınız komutlardır.

Git'de boş bir repository oluşturmak için git init komutunu kullanırız. Init komutunun genel şablonu ve komut ile ilgili yardım almak için kullanabileceğiniz Git komutu aşağıdaki gibidir.

```
git help <komut_adı>
git help init
```

Alıştırma-1

Git Bash veya Terminal komut satırını ara yüzünü açarak herhangi bir parametre kullanmadan aşağıdaki komutları sırası ile çalıştırın

```
Diskinizin C bölümüne konumlanın
cd C:/

# pg_00 isimli boş bir klasör oluşturun. Bu projenizi temsil eden kök klasördür
mkdir pg_00

# pg_00 isimli klasöre konumlanın
cd pg_00

# pg_00 proje klasörünüzü Git deposu haline getirin
git init
```

Alıştırma-2

```
# Diskinizin C bölümüne konumlanın
cd C:/

# pg_00 isimli klasörü silin
rm -r pg_00

# pg_00 proje klasörünü oluşturup Git deposu haline getirin
git init pg_00
```

Alıştırma-3

Bu alışırmaya başlamadan önce kitabın 3. Bölümünde yer alan “Tek Dallı Akış” başlığı altındaki Yalın Depolar ile ilgili açıklamaları tekrar okuyun.

Aşağıdaki komutları sırası ile çalıştırın ve en son komutun verdiği bilgi mesajını bir yere not edin.

```
cd C:/

rm -r pg_00

git init --bare pg_00.git

cd pg_00.git

git status
```

[<< Geri](#) | [İleri >>](#)

3.Gün: Klonlama

Klonlama işlemi ile uzakta bulunan bir Git deposunu kendi bilgisayarımıza indiriyoruz. Uzak depoyu klonlandıktan sonraki tüm değişiklikler artık bilgisayarımızdaki klon Git deposunda kayıt altına alınacak, ta ki git push komutu ile değişikliklerimizi uzaktaki depoya geri yazana kadar.

Kolonlama Hazırlık

- **Adım-1**, 1. Gün, 3. Alıştırma Github, Gitlab veya Bitbucket üzerinde oluşturduğunuz projenin HTTP depo adresini öğrenmemiz gerekiyor. Bunun için kullandığınız Git servisinin sayfasına girip deponuzun ana sayfasındaki “Clone” linkini kullanabilirsiniz.
- **Adım-2**, Aşağıdaki iki komutu çalıştırarak Git'in uzak deponuza HTTP üzerinden bağlanırken kullanacağı kullanıcı adını ve şifremizi geçici bir süre için kaydedeceği store bilgisini tanımlamalıyız

```
# Windows üzerinde şifre'nin cacheleneceği store, wincred
git config --global credential.helper wincred

# Uzak depo servis sağlayıcısı depoları için kullanılacak olan kullanıcı adı
git config --global credential.https://<github.com|gitlab.com|bitbucket.com> <kullanıcı_1_adınız>
```

Wincred ve kullanıcı adımızı tanımladıktan sonra git clone komutu ile Git Bash'den (veya Terminal) klonlama işlemi yapıldığında Git size şifrenizi soracak ve şifre girildikten sonra klonlama işlemi gerçekleştirilecek.

DİKKAT: Klonlama işlemi SourceTree veya Git Eye ile değil Git Bash veya Terminal ile komut satırından yapılacak. clone komutu ile ilgili yardım almak için git help clone komutunu kullanabilirsiniz.

Alıştırma-1

Uzak depodaki projenizi Git Bash (Terminal) kullanarak Git'in komut satırı komutları ile kendi bilgisayarınıza klonlayın ve sonrasında aşağıdaki komutları çalıştırın


```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

git status
git log -n 5
```

Alıştırma-2

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

mkdir src
touch <adınız>.txt

# Aşağıdaki komutları çalıştırmadan önce <adınız>.txt dosyasını açıp içine herhangi bir
metin yazın
git status
git log -n 5
```

Sorular

Soru-1: `git status` komutunun çıktısında verilen bilgiyi nasıl yorumlamamız lazım?

Soru-2: `git log` komutunun çıktısında eklediğiniz dosyaya ilişkin herhangi bir bilgi görüyor musunuz?

[<< Geri](#) | [İleri >>](#)

4.Gün: Değişiklikleri Kaydetme

Bu bölümde bilgisayarınızda oluşturduğunuz yerel bir deponun veya uzaktaki bir deponun bilgisayarınızdaki kopyasında yapacağınız değişiklikleri nasıl versiyon kontrolü altına alacağımız ile ilgili bilgilerimizi tazeleyip alıştırmalar yapacağız.

Projenizi ister yerel bir depoda olsun isterseniz uzak bir depodan klonlamış olsun tüm değişiklikleriniz yerel diskinizde gerçekleşecek ve commitleriniz ile oluşturacağınız tüm versiyonlar git tarafından yerel diskinizdeki .git klasöründeki Git veri tabanında kayıt altına alınıp 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

Takip Edilmeyen (Untracked), bu durumdaki dosyalar versiyon kontrolü altında olmayan veya sizin henüz versiyon kontrolü yapmak için Git'e ekmediğiniz dosyalardır. Bu dosyalardaki değişiklikler siz dosyaları Git'e ekmediğiniz sürece versiyon kontrolüne tabi değildir

Takip Altında (Tracked), bu durumdaki 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 copy) 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 yeni bir versiyon olarak mühürlemeden önce kayıt altında tuttuğunuz veri tabanı olarak tanımlayabiliriz.

git add

Git ile versiyon kontrolü altına aldığınız projenize dosya eklediğinizde, dosya sildiğinizde veya var olan bir dosyanın içeriğini değiştirdiğinizde bu değişiklikler Git tarafından otomatik olarak takip edilmez. Git'in bu dosyaları takip etmesi için git add komutu ile bu dosyaları önce Git'e tanıtmamız gerekir.

Alıştırma - 1

Uzak deponuzdan bilgisayarınıza klonladığınız proje deponuzun src klasörü altındaki gun_04_01.txt isimli bir dosya ekleyip bu dosyanın içine adınızı ve soyadınızı yazın ve komut satırı ara yüzünden aşağıdaki komutları çalıştırın

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

git status
```

Soru: git status komutunun sonucunda yeni eklediğiniz dosya yukarıda bahsedilen iki durumdan hangisindedir?

Alıştırma - 2

Uzak deponuzdan bilgisayarınıza klonladığınız proje deponuzun kök klasörü altında gun_04_02 isimli yeni bir klasör oluşturun ve aşağıdaki komutları komut satırı ara yüzünde çalıştırın

```
cd <proje_klasörü>

git add -A .

git status
```

Soru: git status komutunun sonucunda yeni eklediğiniz klasör *tracked* bir klasör haline geldi mi?

Soru: Boş klasörü tracked hale getirmek için nasıl bir yöntem izlemelisiniz?

Alıştırma - 3

git help add komutu çalıştırarak add komutuna aşağıdaki işlemleri desteklemek için hangi parametrelerin geçilebileceğini araştırıp her bir durum için git add komutunu çalıştırın.

- gun_04_03 isimli klasörün altındaki tüm txt uzantılı dosyalar
- gun_04_03 isimli klasörün altındaki tüm dosyalar

- Proje klasörünüze eklenen tüm dosyalar
- gun_04_03.txt isimli tek bir metin dosya

git checkout

Checkout komutu birden fazla amaçla kullanılabilen komutlardan birisidir. Bunlar

- Bir dosyadaki değişiklikleri iptal etmek
- Çalışma kopyanızdaki (Working Copy) değişiklikleri kaybetmeden projenizin herhangi bir commit'ini incelemek
- Farklı bir dalı çalışma kopyanız olarak aktif hale getirmek

Bu bölümde bu komutun kullanımlarından birincisi ile ilgileniyoruz. Checkout komutu ile değiştirdiğimiz ancak henüz git add ile Staging Area'ya eklememiş dosyalardaki değişiklikleri aşağıdaki komut şablonunun kullanarak iptal edebiliriz.

```
git checkout <dosya_yolu_ve_dosya_adı>
```

Alıştırma - 1

Bu alıştırmaı yapmak için öncelikle proje kök klasörü altındaki src klasörüne txt uzantılı metin bir dosya ekleyip dosyayı commit etmeniz gerekiyor. Bu işlemi aşağıdaki komutlar ile hızlıca yapabilirsiniz.

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

touch <dosya_adı_1>.txt

git add -A .

git commit -m "4. gün, ikinci bölüm 1. Alıştırma hazırlığı"
```

Dosyanızı commit ettikten sonra komut satırı ara yüzünde aşağıdaki komutları çalıştırın

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# <dosya_adı>.txt dosyasının içeriğini değiştirdikten sonra aşağıdaki komutları çalıştırın
git status

git checkout <dosya_adı_1>.txt
```

Alıştırma - 2

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# <dosya_adı_2>.txt isimli yeni bir dosya ekleyin ve içine adınızı soyadınızı yazın
git status

git checkout <dosya_adı_2>.txt
```

Soru: Aldığınız hatayı nasıl yorumlamamız lazım.

Alıştırma - 3

```
# önceki alıştırmada eklediğiniz <dosya_adı_2>.txt dosyasını git add ile Staging Area'
ya ekleyin
# dosyanın içeriğini değiştirin ve kaydedin

$ git checkout <dosya_adı_2>.txt
```

Soru: git checkout işlemi sonrasında .txt dosyasının içeriği Staging Area'ya eklenmeden önceki haline döndü mü? Neden?

git reset

Reset komut ile Staging Area'ya git add ile eklediğimiz ve Git tarafından takip edilen (tracked) dosyaları Staging Area'dan çıkarabiliriz. Komut şu şablonu şöyle

```
git reset HEAD <dosya_adı>
```

Komut şablonundaki parametre değerini vermeden de komutu çalıştırabilirsiniz. Bu durumda Staging Area'daki tüm dosyalar bu alandan çıkarılır ve tekrar takip edilmeyen(untracked) durumuna gelir.

Komut şablonundaki **HEAD** parametre değeri verilmeden de komutu çalıştırabilirsiniz. Bu durumda reset işlemi dosyanızı son commit'de olduğu haline geri getirir.

Alıştırma

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Daha önceki alıştırmalarda eklediğiniz ve commit ettiğiniz herhangi bir dosyanın içeriğini değiştirin
git status

# dosyanız henüz track edilmiyor

# Değiştirdiğiniz txt uzantılı tüm dosyaları Staging Area'ya ekleyin
git add *.txt

git status
# dosyalarınız tracked hale geldi

git reset HEAD <dosya_adı>.txt

git status
# <dosya_adı>.txt isimli dosyanız tekrar untracked halde
```

Soru-1: Dosyanızda yaptığınız değişiklik kayboldu mu?

Soru-2: Dosyanızı değişiklik yapılmadan önceki hale getirmek için hangi komutu kullanmanız lazım? (İPUCU: Son durumda dosyanız untracked durumda yani Staging Area'da değil)

[<< Geri](#) | [İleri >>](#)

5.Gün: Değişiklikleri Versiyon Kontrolüne Alma

Git'in temel yaklaşımlarından birisi de değişikliklerinizin versiyon kontrolü altına alınmaya hazır olduğuna karar verene kadar Git ile herhangi bir etkileşime geçmenize gerek olmamasıdır. Staging Area denilen ara bir alanın varlığı da bu yaklaşım ile uyumludur; birden fazla dosyanızı değiştirip mantıksal olarak anlamlı bütünler halinde Staging Area'da yer almasını sağlayıp daha sonra da bu değişiklikleri yeni versiyon (commit) olarak mühürlileyebilirsiniz

Örneğin a.txt, b.txt ve c.txt isimli üç dosyada değişiklik yaptığınızı düşünelim. Bu değişikliklerden a.txt ve b.txt dosyalarındakilerinin birbiri ile ilişkili olduğunu, c.txt dosyasındaki değişikliğin ise kendi başına anlamlı olduğunu düşünelim. Aşağıdaki Git komutları ile değişikliklerimizi iki commit oluşturacak şekilde mantıksal olarak gruplayabilirsiniz.

```
git add a.txt

git add b.txt

git commit -m "a ve b dosyalarında değişiklik yapıldı"

git add c.txt

git commit -m "c dosyasında değişiklik yapıldı"
```

Dosyalarınızda yaptığınız değişikliklerin versiyonlanması için önce Staging Area'ya eklenmesi daha sonra da git commit ile tüm değişikliklerinizin yeni bir versiyon olarak mühürlenerek Git tarafından kayıt altına alınmasını sağlamalısınız. Bu döngü tüm değişiklikler için geçerli olan edit/stage/commit döngüsü olarak anılır.

Alıştırma - 1

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Boş bir dosya oluşturun
touch <dosya_adı>.txt

# Dosyayı Staging Area'ya ekleyin
git add <dosya_adı>.txt

# Değişikliklerinizi commit edin
git commit -m "<dosya_adı> isimli dosyayı ekledim"
```

Soru-1: git commit komutunu -m parametresi kullanmadan çalıştırırsanız ne olur?

Soru-2: Commit tarihçenizdeki commit mesajlarını tek satır halinde görmek için hangi komutu çalıştırmalısınız?

Alıştırma - 2

Git'in commit mesajları ve diğer metin düzenleme işlemler için varsayılan metin editör uygulaması olarak Windows üzerinde Notepad++, OSX üzerinde Sublime Text, Linux üzerinde de gEdit veya kendi tercih ettiğiniz bir metin editörü kullanabilmesi için gerekli Git ayarlarını yapın

Alıştırma - 3

Git'in her bir commit işlemi için otomatik ürettiği hash değerinin ne olduğunu araştırın.

[<< Geri](#) | [İleri >>](#)

6.Gün: Commit Edilmiş Değişiklikleri İptal Etmek

Bu bölüme kadar değişiklikleri geri almanın iki yöntemini ele aldık, hatırlatmak gerekirse

- Track edilmeyen dosyalardaki (unstaged) değişiklikleri iptal etmek için git checkout
- Track edilen fakat henüz commit edilmemiş (staged) dosyalardaki değişiklikleri iptal etmek için ise git reset HEAD

komutlarını kullanabiliriz. Bu bölümde ise commit edilmiş değişiklikleri git revert komutu ile nasıl geri alabileceğimizi hatırlayıp alıştırmalar yapacağız.

Revert komutunun en önemli özelliği commit edilmiş bir değişikliği otomatik olarak yeni bir commit oluşturarak geri almanızı sağlamasıdır. Revert komutunun bu özelliği sayesinde hata ile yapılan commit'ler de proje tarihçesinde kalmaya devam eder. Özellikle diğer kullanıcılar ile paylaşılan bir depoda çalışıyorsanız ve geri almanız gereken bir commitiniz varsa revert komutunu kullanmamız tavsiye ediyoruz. Ekip arkadaşlarımız bizim yanlılıkla oluşturduğumuz commit'e dayanarak geliştirme yapmış olabilirler bu nedenle commitimizi geri alırken proje tarihçesine bir iz bırakmalı ve ekipteki diğer kişilerin revert işleminden haberdar olmalarını sağlamamız gerekiyor.

Özetlemek gerekirse git revert ile yeni bir commit oluşturularak önceki commit'deki değişiklikleri proje tarihçesinde iz bırakarak geri alabiliyoruz.

Alıştırmaya geçmeden önce örnek proje deponuzda yaptığınız tüm değişiklikleri `git push` komutu ile uzak deponuza yazmanız gerekiyor

Alıştırma

```
# Projenizdeki herhangi bir dosyayı açıp içeriğini değiştirin
# adiniz.txt dosyasını staging area'ya ekleyin
git add -A .

# Değişikliğinizi commit edin
git commit -m "6. Gün alıştırma"

# Yerel deponuzu uzak depo ile senkronize edin
git push
```

Son commit işleminin hatalı olduğunu varsayalım. Bu değişikliği iptal edip bir önceki versiyona geri dönmek için aşağıdaki komutları sırası ile çalıştıralım

```
# Commit tarihçesindeki en son 5 commit kaydının bilgilerini listeleyin
git log -n 5

# Geri dönmek istediğiniz commit'in hash değerini belirledikten sonra

git revert <commit_hash_ilk_7_karakter> --no-edit

# Yerel deponuzu uzak depo ile senkronize edin
git push
```

Soru-1: git push ile değişikliklerinizi uzak depoya göndermemiş olsaydınız geri alma işlemini nasıl yapacaktınız?

Soru-2: Son commit hash değerinin tamamı veya 7 karakterini kullanmak yerine farklı nasıl bir parametre kullanabilirdiniz?

Soru-3: Commit hash değerinin ilk 7 karakterini kullanmak neden yeterli?

Soru-4: git reset ile git revert komutları arasındaki fark nedir?

[<< Geri](#) | [İleri >>](#)

7.Gün: Commit Edilmiş Değişiklikleri Silmek

Bu bölüme kadar değişiklikleri geri almanın üç yöntemini ele aldık, hatırlatmak gerekirse

- Track edilmeyen dosyalardaki (unstaged) değişiklikleri iptal etmek için **git checkout [dosya_adi]**
- Track edilen fakat henüz commit edilmemiş (staged) dosyalardaki değişiklikleri iptal etmek için **git reset HEAD [dosya_adi]**
- Commit edilmiş bir değişikliği proje tarihçesini bozmadan **git revert [commit_hash]**

komutlarını kullanabiliriz.

Bu bölümde ise commit edilmiş değişiklikleri `git reset --hard` komutu ile proje tarihçesinden nasıl silebileceğimizi ele alıyoruz. Reset komutunu **--hard** parametresi ile paylaşımlı çalışılan projelerde özellikle uzak depoya yazdığınız değişiklikler için kullanmamanızı öneriyorum. Ancak, çalışma kopyanızda (Working Copy) henüz uzak depoya yazarak ekibinizin geri kalanı ile paylaşmadığınız commitleri **--hard** parametresi ile silebilirsiniz.

Özetlemek gerekirse `git reset --hard <commit_hash>` komutu ile yeni bir commit oluşturmadan geri dönülmez bir şekilde herhangi bir commit'i silebilirsiniz.

Alıştırma-1: Tag oluşturma

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Yeni bir dosya oluşturun
touch <dosya_adi>.txt

# Oluşturduğunuz dosyayı commit edelim
git add -A .
git commit -m "7. gün 1. alıştırma"

# Deponuzun şu anki halini yansıtan v0.1 isimli tag oluşturduk
git tag v0.1
# Depomuzdaki tag'ları listeledik
git tag

# Depomuzdaki değişiklikleri uzak depo ile senkronize ettik
git push
```

Alıştırma-2: Değişikliğin geri alınması

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# 1. Alıştırmadaki dosyanızın içeriğini değiştirin

# Dosyadaki değişiklikleri commit edelim
git add -A .
git commit -m "7. gün 2. alıştırma"

# Depomuzu v0.1 ile tagladığımız haline geri döndürelim.
git reset --hard v0.1

# Oluşturduğunuz dosyanın içeriğinin v0.1'deki hali ile aynı olup olmadığını kontrol e
din

# Depomuzun commit tarihçesindeki son 5 commiti listeleyelim
git log -n 5

# Depomuzdaki değişiklikleri uzak depo ile senkronize ettik
git push
```

Soru-1: Dosyanızın içeriğini tekrar değiştirerek add/commit/push işlemlerini yapın. Sonra projenizi git reset --hard v0.1 komutu ile tekrar v0.1 versiyonuna geri alıp ardından git push ile uzak depoya göndermeye çalıştığınızda nasıl bir hata ile karşılaşıyorsunuz? Bu durumda v0.1'e geri dönmek için ne yapmanız lazım?

Soru-2: Oluşturduğunuz v0.1 isimli tag'ı nasıl silersiniz?

Soru-3: Oluşturduğunuz tag'ları uzak deponuza nasıl gönderebilirsiniz?

[<< Geri](#) | [İleri >>](#)

8. Gün: Dal Oluşturmak

Projelerimizde önemli ve göreceli olarak uzun sürebilecek değişikliklere başlamadan önce yerel deponuzda bir dal (branch) oluşturup değişiklikleri bu dal üzerinde yapmanızı öneriyorum. Bu öneri aslında Konu Dalları Akışı adı verilen ve kitabımızın 3. Bölümünde ele aldığımız Git iş akışlarından bir tanesini tanımlar.

Oluşturduğunuz yerel dal üzerinden çalışmalarınızı yapmaya devam ederken herhangi bir anda, örneğin acil bir hata durumunu gidermek için master (master olmak zorunda değil başka herhangi bir dal de olabilir) dalı aktif hale getirip oradaki düzenlememizi yaparak tekrar önceki dal'a geri dönebiliriz. Bu geçişler sırasında dallarda yaptığınız değişiklikler korunur.

Git'de dal oluşturmak için iki yöntemden birini kullanabilirsiniz

Yöntem-1, branch komutu

```
git branch <dalınızın adı>
```

Branch komutu ile dalınızı oluşturduktan sonra yeni dalda çalışmaya başlayabilmek için `git checkout <dalınızın adı>` komutu ile dalınızı aktif hale getirmeniz lazım.

Yöntem-2, checkout komutu

```
git checkout -b <dalınızın adı>
```

Bu komut ile yeni bir dal oluşturulur ve otomatik olarak yeni dal için checkout işlemi yapılır.

DİKKAT: Yeni dal oluşturma komutlarını yazdığınız sırada hangi dalın aktif olduğu önemlidir. Her iki komut da yeni dalı o sırada aktif olan dalı referans alarak aktif dalın bir kopyasını oluşturur.

Dal İşlemleri

```
# Tüm dalları listele
git branch

# Bir dalı silme
git branch -d <dalinızın adı>

# Dalın adını değiştir
git branch -m
```

İPUCU: git status komutunu çalıştırdığınızda aktif olan dalın adını da görebilirsiniz

Alıştırma-1

```
# yöntem 1
git branch <dalinızın adı>
git branch -d <dalinızın adı>

$ # yöntem 2
$ git checkout -b <adiniz>
$ git branch -d <adiniz>
```

Soru-1: Yukarıdaki alıştırmada kullandığımız git branch komutlarının çıktıları arasında nasıl bir fark var?

Soru-2: 2. yöntem ile oluşturduğunuz dalı nasıl silersiniz?

Alıştırma-2

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Yeni bir dal oluşturun
git checkout -b <dalinızın adı>
# Proje klasörünüze yeni bir dosya ekleyin
# Eklediğiniz dosyayı commit edin

# master dalını aktif hale getirin
git checkout master

# Dalı silin
git branch -d <dalinızın adı>
```

Alıştırma-3

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Yeni bir dal oluşturun
git checkout -b <dalınızın adı>
# Proje klasörünüze yeni bir dosya ekleyin

# master dalını aktif hale getirin
git checkout master

# Dalı silin
git branch -d <dalınızın adı>

# Deponuzun durumunu inceleyin
git status
```

Soru-1: En son çalıştırdığınız git status komutu eklediğiniz dosyayı neden listeliyor?

Alıştırma-4

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Yeni bir dosya oluşturun
touch <dosyanızın adı>.txt

# Yeni bir dosya oluşturun
git add -A .
git commit -m "8.gün 4. alıştırma"

# Yeni dosyanızın içeriğini metin editörü ile değiştirin ve sonra aşağıdaki komutu çalıştırın

git stash

# Yeni dosyanızın metin editörü ile açıp içeriğini kontrol edin
```

Soru-1: stash komutunu çalıştırmadan önceki dosya içeriğinizi nasıl geri getirebilirsiniz?

[<< Geri](#) | [İleri >>](#)

9.Gün : Değişiklikleri Birleştirme (Merge)

Oluşturduğumuz dallardaki değişiklikleri master dalına veya master dalındaki değişiklikleri üzerinde çalıştığınız başka bir dala entegre işlemine birleştirme (merge) denir.

Alıştırmalarımıza geçmeden önce aşağıdaki komutları kullanarak git log komutu için bir alias (kısaltma) oluşturalım.

```
git config --global alias.hist = "log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short"
```

Alıştırma-1

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Yeni bir dal oluşturalım
git checkout -b <dalınızın adı>

# Yeni bir dosya oluşturalım
touch dosya1.txt
# Oluşturduğunuz dosyayı commit edin
git add -A .
git commit -m "Dosya 1 eklendi"

# Yeni bir dosya daha oluşturalım
touch dosya2.txt
# Oluşturduğunuz dosyayı commit edin
git add -A .
git commit -m "Dosya 2 eklendi"

# Yeni bir dosya daha oluşturalım
touch dosya3.txt
# Oluşturduğunuz dosyayı commit edin
git add -A .
git commit -m "Dosya 3 eklendi"
```

Alıştırma-2


```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# master dalını aktif hale getirelim
$ git checkout master

# Proje klasörünüzün içeriğini listeleyin
ls -lah

# Yeni bir dosya oluşturalım
touch ornek1.txt

# Oluşturduğunuz dosyayı commit edin
git add -A .
git commit -m "Örnek 1 eklendi"
```

Soru-1: git hist --all komutu çalıştırdığınızda iki alıştırmada yaptığınız işlemleri görüyor musunuz? Çıktıyı nasıl yorumlarsınız?

Alıştırma-3

master dalındaki değişiklikleri oluşturduğunuz dal'a entegre edelim. Bu durumda birleştirme işlemi için master kaynak oluşturduğunuz diğer dal da hedeftir.

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Oluşturduğunuz dalı aktif hale getirin
git checkout <dalınızın adı>

# master dalındaki değişiklikleri birleştirin
git merge master

git hist --all
```

Alıştırma-4

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# master dalını aktif hale getirelim
git checkout master

# Eklediğiniz ornek1.txt dosyasını metin editörünüz ile açıp içine adınızı ve soyadınızı yazın
# Değişikliklerinizi commit edin
git add -A .
git commit -m "Ad ve soyad eklendi"

# Oluşturulan dalı aktif hale getirelim
git checkout <dalınızın adı>

# ornek1.txt dosyasını metin editörünüz ile açıp içine Deneme yazın
# Değişikliklerinizi commit edin
git add -A .
git commit -m "Ad soyad bilgisi Deneme olarak değiştirildi"

git hist --all
git merge master
```

Soru-1: Çakışma oluşan dosyayı açın ve çakışma olan satırların nasıl gösterildiğini inceleyip gösterimde kullanılan notasyonu yorumlayın.

Soru-2: git mergetool komutunu çalıştırın ve bu komutun ne işe yaradığını yorumlayın.

Soru-3: merge/diff işlemlerinde Windows üzerinde WinMerge uygulamasının kullanılması için Git’de nasıl bir konfigürasyon yapmanız gerektiğini araştırın. (OSX kullanıyorsanız SourceGear DiffMerge, Linux kullanıyorsanız da KDiff3 için araştırmanızı yapabilirsiniz)

[<< Geri](#) | [İleri >>](#)

10.Gün: Rebase

Git’de merge ve rebase komutları benzer işlevleri yerine getirmek için kullanılıyor. Her iki komut da bir daldaki değişiklikleri başka bir dala birleştirmek için kullanılır. Ancak bu iki komut arasında proje tarihçesinin oluşturulması ile ilgili ciddi bir farklılık vardır.

Merge komutu ile A dalındaki değişiklikler B dalı ile birleştirildiğinde B dalının commit tarihçesinde merge işleminden kaynaklanan ve merge commit adı verilen otomatik oluşturulmuş bir commit yer alır. Bu commit A ve B dallarının tarihçelerini birbiri ile ilişkilendirir.

Merge commit’in ayrıntılarını kitabımızın 5. Bölümünde “Merge Alternatifi Olarak Rebase Kullanımı” başlığı altında ele almıştık.

rebase komutu kullandığımızda ise A dalındaki her bir commit B dalına sanki commit işlemi B dalında yapılmış gibi yeniden yazılır. Bu sayede B dalının commit tarihçesi sanki tüm değişiklikler bu dalda olmuş gibi düz ve kesintisiz görünür.

Ne zaman rebase kullanmalıyız?

Rebase komutu yerel ve kısa süreliğine (örneğin bir hata giderme veya deneysel bir çalışma için oluşturulan) geçerli dallar için kullanılmalı. Paylaşılan depolarda rebase komutunu kullanmamanızı tavsiye ediyorum, çünkü rebase sonrasında projenizin ana dallarında değişikliklerin nereden kaynaklandığına dair bir bilgi veya ipucu göremezsiniz. Bu durum takım çalışmasını olumsuz yönde etkiler.

Alıştırmalara başlamadan önce master dalından kopyalanan yerel bir dal oluşturun

Alıştırma-1

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# master dalını aktif hale getirelim
git checkout master

# Yeni bir dosya oluşturun
touch ornek_rebase1.txt

# Değişikliklerinizi commit edin
git add -A .
git commit -m "Ad ve soyad eklendi"
```

Alıştırma-2

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Oluşturduğunuz dalı aktif hale getirelim
git checkout <dalınızın adı>

# rebase işlemini yapın
git rebase master

# dalınızın commit tarihçesinin bilgilerini listeleyen
git hist --all # repository tarihçesine göz atın
```

Soru-1: Merge ile rebase arasındaki farkı yorumlayın.

Alıştırma-3

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# master dalını aktif hale getirelim
git checkout master

# ornek_rebase1 dosyasının içeriğini metin editörünüz ile değiştirin
git add -A .
git commit -m "Master dalındaki değişiklik mesajınız"

# Dalınızı aktif hale getirin
git checkout <dalınızın adı>

# ornek_rebase1 dosyasının içeriğini metin editörünüz ile değiştirin
git add -A .
git commit -m "Diğer daldaki değişiklik mesajınız"

# rebase işlemini yapın
git rebase master
```

[<< Geri](#) | [İleri >>](#)

11.Gün: Birlikte Çalışma (Collaboration)

Bu bölüme kadar ele aldığımız Git komutları ile kendi lokalimizde Git ile temel versiyon kontrol işlemlerini nasıl yapacağımız ile ilgili alıştırmalar yaptık. Bu bölümde ise takım çalışması için, tüm diğer versiyon kontrol sistemlerinde olduğu gibi, Git'in de bize sunduğu imkanlar ile ilgili alıştırmalar yapacağız .

Takım çalışması ve birlikte çalışma modeli açısından Git'i öncülü olan CVS ve Subversion gibi versiyon kontrol sistemlerinden ayıran en önemli iki özelliği şunlardır

- Git her takım üyesine merkezi uzak deponun tam bir yerel kopyasını sağlar. Takım üyeleri bu yerel kopya üzerinde kendi commit'lerini ve dallarını merkezi depo'ya ihtiyaç duymadan istedikleri gibi oluşturup yönetebilirler
- Git ile takım üyeleri değişiklik kümeleri yerine (change set) tekil değişiklikleri ifade eden commit'leri birbirleri ile merkezi depo üzerinden paylaşırlar.

Bu bölümde ele aldığımız komutlar ile diğer repository'ler ile olan bağlantıların nasıl sağlandığını (remote), kendi lokalimizdeki değişiklikleri diğerleri ile nasıl paylaşacağımızı (push) ve diğerlerinin değişikliklerini kendi lokalimize nasıl entegre edeceğimizi (pull) öğreneceğiz.

git remote

Remote komutu ile yerel deponuz ile uzaktaki bir depo arasında bağlantıları tanımlayabilir, silebilir veya isimlerini değiştirebilirsiniz. Remote komutu ile tanımlanan bağlantı gerçek zamanlı ve canlı bir bağlantı değildir ve bu bağlantılar kullanılarak uzaktaki depo üzerinden doğrudan işlem yapamazsınız. Bu bağlantıları uzun URL'ler yerine diğer Git komutlarında parametre olarak kullanılabilecek kısa yollar olarak düşünmelisiniz.

```
# Proje klasörünüze konumlanın
cd <proje_kök_klasörü_yolu>

# Uzak depo kısa yol isimlerini listele
git remote

# Uzak depo ilişkilerini daha ayrıntılı listele
git remote -v

# Uzak depo ilişkisi tanımla
git remote add <kısa yol adı> <uzak depo adresi>

# Uzak depo ilişkisini sil
git remote rm <kısa yol adı>

# Uzak depo kısa yolunun adını değiştir
git remote rename <kısa yol adı> <kısa yol yeni adı>
```

Soru-1: remote komutunu çalıştırdığınızda çıktı olarak gördüğünüz origin ne anlama geliyor?

Soru-2: git remote add komutu ile kullanabileceğimiz URL tipleri nelerdir?

git fetch

Fetch komutu uzak depolardaki değişiklikler ile ilgili bilgeleri yerel deponuzun Git veri tabanı ile senkronize etmenizi sağlar.

```
# varsayılan uzak depodaki (origin) tüm dalların bilgilerini almak için
git fetch

# Uzak depo kısa yolu ile tanımlı uzak depodaki tüm dalların bilgilerini almak için
git fetch <uzak depo kısa yolu>

# Uzak depo kısa yolu ile tanımlı uzak depodaki belirli bir dalın bilgilerini almak için
git fetch <uzak depo kısa yolu> <dal adı>
```

Fetch sonrasında uzak depo dallarını git checkout ile aktif hale getirip inceleyebilir ve git log ile de uzak depo dalının commit tarihçesine göz atabilirsiniz. Eğer uzak depo dalındaki değişiklikleri kendi yerel deponuzdaki bir dala birleştirmek etmek isterseniz git merge komutunu kullanabilirsiniz.

Alıştırma

Projeniz için Git servisi üzerinde oluşturduğunuz uzak depodaki değişiklikleri fetch ile alıp yerel dalınız ile birleştirin. Bu alıştırmayı yapmak için uzak deponuzu bu aşamaya kadar kullandığınız klasörden farklı bir konuma klonlayın ve bu klon üzerinden değişikli yaparak uzak deponuza push edin.

Soru-1: downstream ve upstream terimlerinin ne anlama geldiğini araştırın.

git pull

Uzak depodaki değişikliklerin yerel deponuza kopyalanması ve otomatik olarak entegre edilmesi için kullanılır. Bu işlemi git fetch ve sonrasında git merge ile iki komut kullanarak da yapabilirsiniz, ancak git pull bu işlemi tek komutla yapmamızı sağlar.

```
# varsayılan uzak depodaki (origin) değişiklikleri almak için
git pull

# Uzak depo kısa yolu ile ilişkilendirilmiş konumdan değişiklikleri almak için
git pull <uzak depo kısa yolu>

# Merge yerine rebase kullanarak değişiklikleri almak için
git pull -rebase <uzak depo kısa yolu>
```

git push

Yerel deponuzdaki commit'leri (değişiklikleri) uzaktaki depoya gönderip takımın geri kalanı ile paylaşmak için git push komutu kullanılır.

```
# Yerel daldaki değişiklikleri uzak depoya gönder
git push

# Bir daldaki değişiklikleri uzak depo kısa yolu ile ilişkilendirilmiş uzak depoya gönder
git push <uzak depo kısa yolu> <dalın adı>

# Tüm yerel dallardaki değişiklikleri uzak depo kısa yolu ile ilişkilendirilmiş uzak depoya gönder
git push <uzak depo kısa yolu> -- all

# Yerel depodaki tag'leri uzak depoya gönder
git push <uzak depo kısa yolu> -- tags
```

Soru-1: push işlemi sırasında bir çakışma oluşabilir mi?

[<< Geri](#) | [Ana Sayfa](#)

