Git Versiyon Kontrol Sistemine Giriş

Table of Contents

1. Introduction
2. Versiyon Kontrolüne Giriş
    i. Versiyon Kontrolü Nedir?
    ii. Versiyon Kontrolüne Neden İhtiyacımız Var?
    iii. Kısa Git Tarihçesi
    iv. Git İle Çalışmaya Başlamak
    v. Basit Anlamda Versiyon Kontrolü İş Akışı
    vi. Local bir proje oluşturmak
    vii. Remote bir proje oluşturmak
    viii. Projemiz Üzerinde Çalışmaya Başlayalım
3. Branching (Dallanma) ve Merging (Birleştirme)
    i. Branching Çalışma Şeklinizi Değiştirebilir
    ii. Branch'ler İle Çalışmak
    iii. Değişikliklerinizi Geçici Olarak Kaydetmek -> Git Stash
    iv. Local Bir Branch'de Çalışmak
    v. Değişiklikleri Merge Etmek
    vi. Branching İş Akışları
4. Remote Repository'ler
    i. Remote Bir Repository'ye Bağlantı Sağlamak
    ii. Remote Repository'deki Verilerin İncelenmesi
    iii. Remote Değişiklikleri Entegre Etmek
    iv. Local Bir Branch'i Yayınlamak (Publish)
    v. Branch'leri Silmek
5. İleri Seviye Komutlar ve İşlemler
    i. Değişikliklerinizi Geri Almak
    ii. Diff İle Farkları İncelemek
    iii. Çakışmaları Gidermek
    iv. Merge Alternatifi Olarak Rebase Kullanımı
6. Git Araç ve Servisleri
    i. Görsel Git İstemcileri
    ii. Diff/Merge Araçları
    iii. Git Servisleri
    iv. Kaynakça ve Referanslar

Türkçe Git 101

Önsöz

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.

İngilizce Terimler

Yazılım Geliştirme ile ilgili çoğu konuda olduğu gibi maalesef Git ile ilgili kaynaklar da ağırlıklı olarak İngilizce. Terminoloji anlamında Türkçe bir kaynak hazırlamanın en büyük zorluğu İngilizce terimlere Türkçe uygun karşılık bulmaktır. Ancak **Git 101** kitabında İnglizce -> Türkçe geçişini birebir yapmayacağım, mümkün olduğu kadar Versiyon Kontrolü ve Git ile ilgili terimlerin İnglizce hallerini kullanmaya çalışacağım.

Konuların diziliminde ve içeriğin oluşturulmasında Learn Version Control with Git isimli kitapçığın online versiyonu baz alınmıştır. Belirtilen kaynaktaki başlıklara ve içeriğe ilave olarak daha ayrıntılı bir kitap olan Pro Git kitabından ve son bölümde linklerini paylaştığım online diğer kaynaklardan da faydalanılmıştır.

Örnekler

Örneklerimizi **Terminal** (komut satırı veya command line olarak da tabir edilen) üzerinden Apple Mac OS X işletim sistemi kullanarak oluşturacağız. Bu kaynağın oluşturulduğu anda benim bilgisayarımdaki Apple Mac OS X ve git versiyonları aşağıdaki gibi

- OS X versiyonu : 10.9.4 (Mavericks)
- Git versiyonu : 1.8.5.2 (Apple Git-48)

Git, OS X'in yanısıra tüm Linux dağıtımları ve Windows'da da çalışmaktadır. Git komutları kullandığınız işletim sistemine göre değişmez ancak Git kurulumu kullandığınız işletim sistemine göre değişebilir. İşletim sisteminize bağlı olarak kurulum yönergeleri için arama motorlarını kullanarak gerekli adımları öğrenebilirsiniz.

Versiyon Kontrolüne Giriş

Bu bölümde aşağıdaki konuları ele alacağız

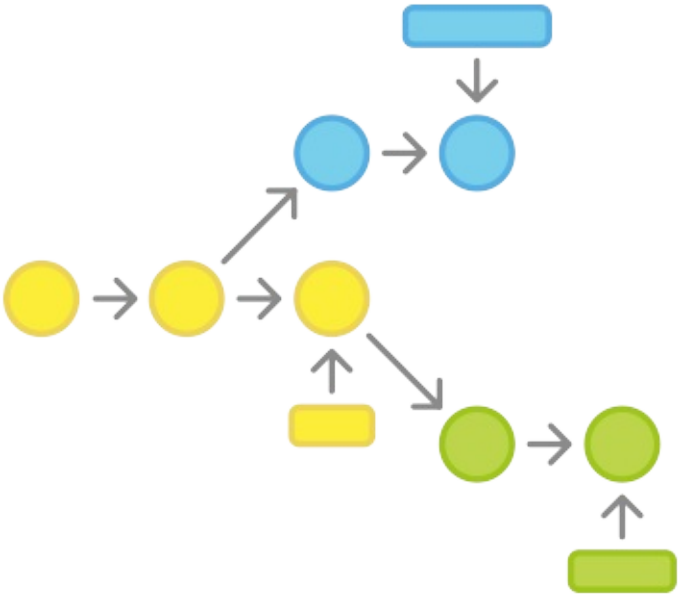
- Versiyon Kontrolü Nedir
- Versiyon Kontrolüne Neden İhtiyacımız Var
- Git İle Çalışmaya Başlamak
- Basit Anlamda Bir Versiyon Kontrolü İş Akışı
- Local Bir Proje Oluşturmak
- Remote Bir Proje Oluşturmak
- Projemiz Üzerinde Çalışmaya Başlayalım

Versiyon Kontrolü Nedir?

Versiyon kontrolü nedir ve bizi neden ilgilendirmeli? Versiyon kontrolünü bir dosya veya bir küme dosyadaki değişiklikleri takip edebilmek için uyguladığımız bir yöntem olarak tanımlayabiliriz. Git gibi sistemler tüm bu değişikliklerin tarihçesini ve içeriğini elektronik olarak bizim için takip ederek kayıt altına almamızı sağlayan veritabanları olarak düşünülebilir.

Bu sistemleri kullanarak herhangi bir anda üzerinde çalıştığınız dosyaların o anki hallerini kaydedebilir, daha sonra da isterseniz bu dosyaların kaydedilmiş ve kontrol altına alınmış herhangi bir haline geri dönebilirsiniz.

> Dosyaların kayıt altına alınmış herhangi bir andaki hallerine **versiyon** diyoruz



Görsel : Atlassian Git Workflows sayfasından alıntı

Versiyon kontrolünü, kullandığınız programlama dili, yardımcı programlama kütüphaneleri (framework), dosya tipi veya işletim sisteminden bağımsız bir yaklaşım olarak düşünmelisiniz. Çünkü versiyon kontrolü

- HTML dosyalar için kullanılabileceği gibi, mimari tasarım amaçlı proje dosyaları ve iPhone uygulaması kaynak kodunuz için de kullanılabilir
- Dosyalarınız üzerinde çalışırken hangi işletim sistemini veya hangi programları kullandığınız ile ilgilenmez (Sublime Text, Notepad, Visual Studio, Word, AutoCAD)

> Ben de bu kitabın versiyon kontrolü için **Git** kullanıyorum

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

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 ağırısına 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 kotrol 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 commentler 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 proje'yi sunucuya geri yüklemesi yeterlidir. Diğerleri de kendi bilgisayarlarındaki proje dosylarını 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 miliyolarca 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 ilgli 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'in 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 çalıştırısanız 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 ilgli ilave bilgileri (değişikliği kim, ne zaman yaptı ve değişiklik ile ilgili girilen açıklamalar) ayrı birer **versiyon** olarak kayıt altında tutan bir vertabanıdır. 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 kontrolüde olmayan bir projeniz varsa **git init* komutu ile projenizi tüm klasör ve dosyaları ile birlikte versiyon kontrolüne alabilirsiniz
- Projeniz uzaktaki veya şirket ağınızdaki bir Git sunucusunda versiyon kontrolü altında tutuluyorsa projeyi kendi bilgisayarınıza **git clone** komutu ile indirebilirsiniz.

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

1. Projenizin repositorysini 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 alırız.

   > 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 repositoryde yayınlamanız gerekir. Bunun için *git push* komutunu kullanırız.

> **Local (Yerel) & Remote (Uzak) Repository'ler**
>
> - Local repository, kendi bilgisyarınızda 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'in yayınladığı derlemeye göz atabilirsiniz.

Versiyon kontrolü altına almak istemediğiniz dosya ve klasörleri tanımlamak için proje klasörüne eklenen *.gitignore* dosyası kullanılır. Bu dosya'ya 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'in 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 tutması 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 **/** karekteri kullanılır. Örneğin **/projemde/versiyon/kontrolü/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 commitimiz

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 komut'u 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 internet'de 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 gerken tek şey bu adresten projenizin dosyalarını yerel diskinize klonlmak. Bunun için öncelikle yerel diskinizde projenizi indireceğiniz bir klasör oluşturmanız ve Terminal'den bu klasöre gitemeniz 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 oluştruduğumuza 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 **commitleriniz** 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 eklemediğiniz dosyalardır. Bu dosyalardaki değişiklikler siz dosyaları git'e eklemediğiniz sürece versiyon kontrolüne tabi değildir
- Tracked (Takip Edilen): Bu dosyalar ise git'in versiyon kontrolü takibi altında olan dosyalardır. Bu dosyalar üzerinde yapacağınız tüm değişiklikler git tarafından takip edilmektedir.

Staging Area

Çoğu versiyon kontrol sisteminde değişiklikleriniz iki yerde kaydedilir

- Yerel diskinizdeki çalışma klasörünüz (working folder) veya
- Versiyon kontrol sisteminin veritabanı

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 veritabanı/alan olarak tanımlayabiliriz.

> Versyon 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
>
> - Commitinizdeki 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 commitler 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 eklemediğimzi dosyalardır. Bu dosyalar bir önceki grubun içine eklemediğimiz sürece bir sonraki commit'e dahil olmayacaklarıdır
- Utracked files (Versiyon takbinide olmayan dosyalar): Bu gruptaki dosyalar ise henüz versiyon kontrolü altına almadığımız dosyalardır.

"git add" ve "git rm" komutları

Bir öncelki 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 aştındaki 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 commitimiz'de 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 editör'e mesajınızı istediğimiz uzunlukta girebilirsiniz.

> Versyon Kontrolünün Altın Kuralları
>
> **#2 Anlamlı Commit Mesajları**
>
> Commit işlemi sırasında yazacağınız bilgilendirici bir mesaj hem ekibinizdeki diğer kişilerin hem de daha sonra kendinzin yapılan değişikliği daha rahat ve hızlı anlamanızı 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 isytiyorsanız 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 getirebilirseniz 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 commitleriniz ile ilgili kayıt altına aldığı tarihsel bilgileri görmek için **git log** komutunu kullanıyoruz. BU komut tüm commitler ile ilgili bilgileri, en son commit en üstte olacak şekilde, tarihsel olarak sıralar. Eğer Terminal pencerenize sığımayacak kadar çok tarihsel kayıt var ise son satırda **:** simgesi yer alacaktır, klavyenizden **SPACE/BOŞLUK** tuşuna basarak bir sonraki sayfanın listelenmesini **q** tuşuna basarak da listelemenin sonlandırılmasını sağlayabilirsiniz.

```
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 anumaralı 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 yanısıra, 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 değişikliklerniz ve commit'in kendisi ile ilgili bilgiler de kullanılarak otomatik hesaplanır. Genel olarak git'in listelemelerinde ve bazı komutların parametresi olarak bu değerin ilk 7 karakterinin kullanılması yeterlidir. Çünkü bu ilk 7 karakterin de nerdeyse benzersiz ve tekil olduğunu söyleyebiliriz.

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

```
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
 * İnglizce->Türkçe geçişinde terimlerin anlamını yitirmesine neden olmadan
 * Uluslar arası ekipler ve özellikle açık kaynak projelerde ortak dil İnglizce 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 (Birleştirme)

Bu bölümümüzde aşağıdaki konuları ele alacağız

- Branching Çalışma Şeklinizi Değiştirebilir
- Branch'ler İle Çalışmak
- Değişikliklerinizi Geçici Olarak Kaydetmek -> Git Stash
- Basit Bir Branching Akışı
- Değişiklikleri 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 (Türkçe'ye **dallanma** olarak da çeviebiliriz) da bahsettiğim bu dönüştürücü etkiye sahip araçlardan birisidir. Branching konusundaki hakimiyetimizin artması ve sağlamlaşması ile birliktedaha 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 kendinde has olduğunu da söylemeliyiz.

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

Birden Fazla Bağlamda Çalışmak

Daha önceki bölüm'ün 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 üzeirnde 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 altermatif 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 alış veriş 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 çöp atacaktınız?

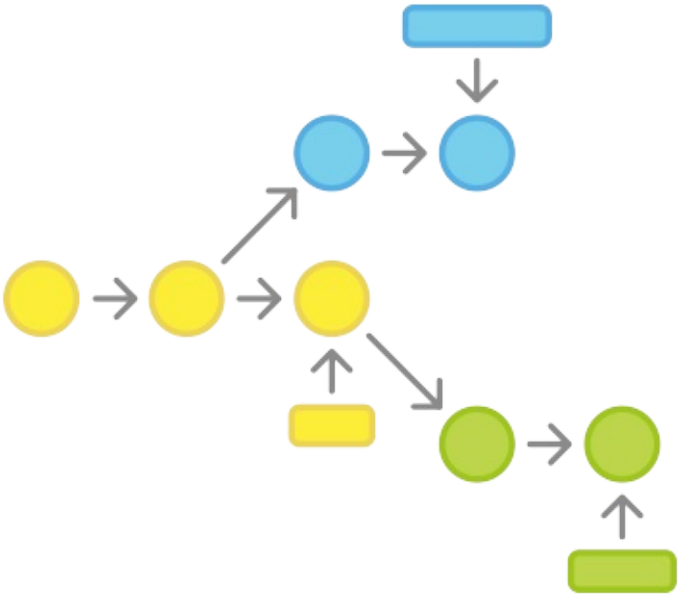
Birden fazla konu ile ilgili değişikliklerin tamamını tek bir bağlam ile yönetmeye çalışırsanız işler hızla sarpa saracaktır. Bu karmaşanın önüne geçmek için her bir değişiklik için projenizin tamamının farklı klasörlere kopyalamayı deneyebilirsiniz. Ancak bu durumda

- 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 kısası 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 branchler 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.

Versyon Kontrolünün Altın Kuralları

#3 Branch'leri Bol Bol Kullanın

Branchler 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 veilen 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 başınada ***** 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 gösterilir
>
> ```
> Alis-MacBook-Pro-2:git101 aliozgur$ git status
> On branch master
> nothing to commit, working directory clean
> Alis-MacBook-Pro-2:git101 aliozgur$
> ```

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?

Versyon 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 yağtığınız değişiklikleri kayıt altına almak için henüz tamamlanmamış değişiklikleri commit etmek zorunda kalmazsınız. Buna rağmen ara safhada kayıt altına almak istediğiniz değişikliker 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 ilgilienmeye başlamak için önceki değişikliklerinizi kaybetmeden yeni ve temiz bir branch oluşturmalısınız. 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$
```

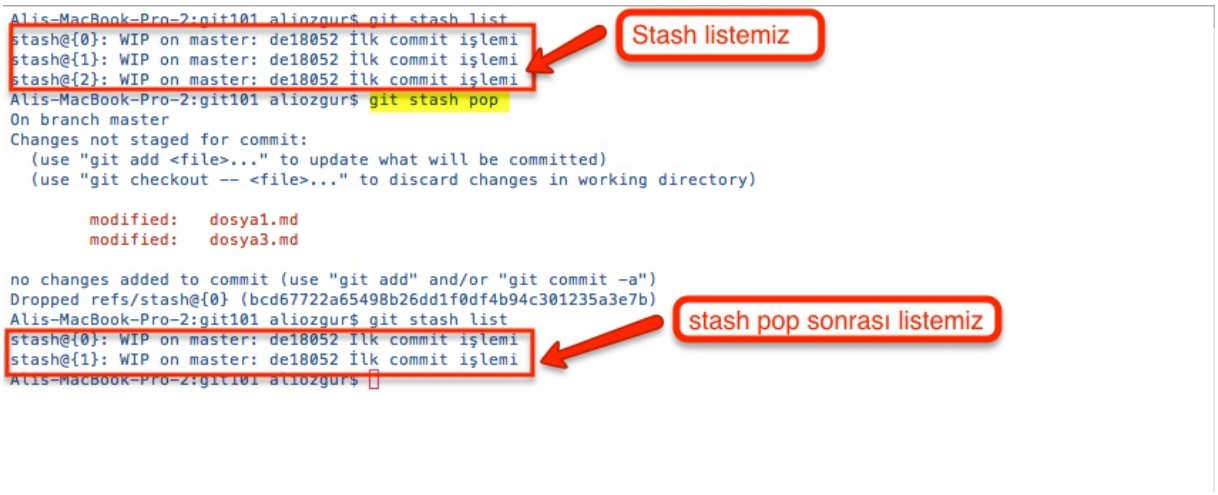
git stash ile üzerinde çalıştığınız ancak henüz commit etmediğiniz değişikliklerin geçici olarak Git tarafından kayıt altına alınmasını ve aktif branch'inizin herhangi bir değişikliğin olmadığı temiz bir duruma getirilmesini sağlarsınız. **git stash** komutunu çalıştırdıktan sonra tekrar **git status**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

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

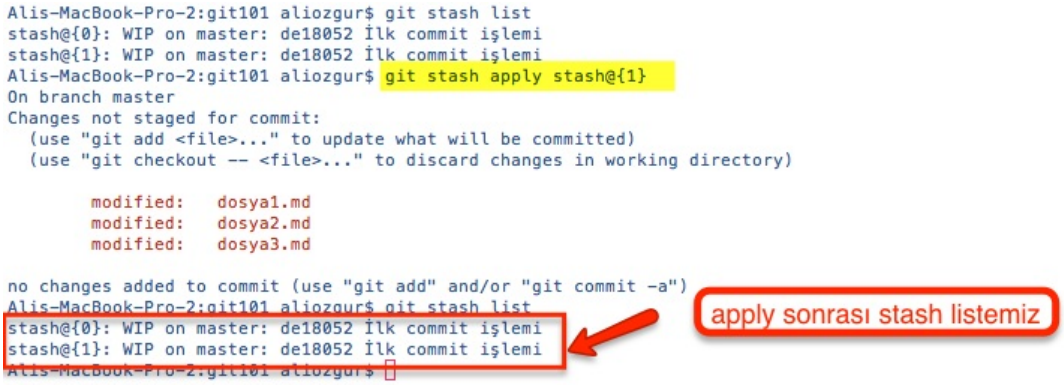
```
Alis-MacBook-Pro-2:git101 aliozgur$ git stash list
stash@{0}: WIP on master: de18052 İlk commit işlemi
stash@{1}: WIP on master: de18052 İlk commit işlemi
stash@{2}: WIP on master: de18052 İlk commit işlemi
Alis-MacBook-Pro-2:git101 aliozgur$
```

Yukarıda görünen lsitede 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

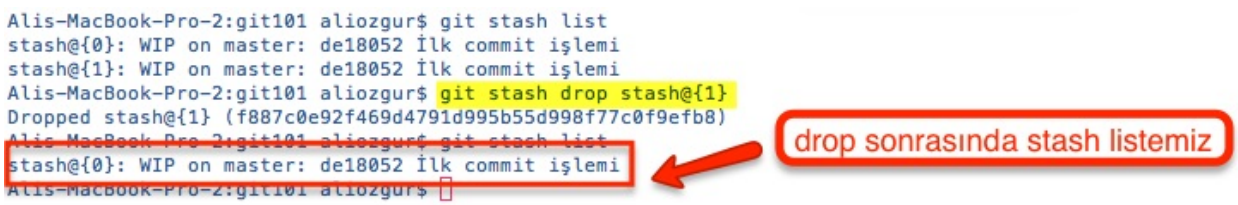
- **git stash pop** komutu ile yukarıdaki listenin en üstünde yer alan değişiklik geri yüklenecek ve bu değişiklik listeden silinecek.



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



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



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 versiyonun bulunduğu **master** branch olarak değiştirdiniz

   > **git checkout master** komutunu kullandık

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

   > - **git branch loginsorunu** koutunu 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** branchimizi aktif hale getirdik

   > **git checkout master** komutu ile

5. Commit ettiğiniz değişikliği web sitenizin stabil versiyonunu içeren **master** branchimize 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 branchlerinizdeki dosyalar disikiniz üzerinde değil Git'in veritabanında (.git klasörü için 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 veritabanınızdan 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 işlermlerimizi 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** branchiniz ile merge etmek zorunda değilsiniz. Kullandığınız Git çalışma pratiğine bağlı olarak herhangi bir branch'i başka bir branch'e merge edebilirsiniz.

Değişikliklerinizi **master** branchinize 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 ettğiniz 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 istedğinizi teker söylemezsiniz. Bunun yerine Git'de dorğrudan 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 değişiklikleri HEAD'e entegre edilir

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

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 ve hedeh 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 gruplama sadece anlamsal ve kullanım pratikleri ile ilgili bir gruplandırmadır, sonuç itibariyle branch kavramı daha önceki bölümlerde anlattığımız kadar basit bir Git aracıdır

Kısa Vadeli / Konu Bazlı Branch'ler

Daha önceki bölümlerde branch kullanımı noktasında elinizi korkaka alıştırmamanız ile ilgili tavsiyelerde bulunduk. Örneğin yeni özellikleri 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 branchler projeniz üzerinde geliştirme yaptığınız sürece varlıklarını sürdüreceklerdir. Tipik olarak bu tür branch'ler ile ilgili aşağıdaki kurallar geçerlidir

- Genelde bu tür uzun soluklu branchler üzerinde doğrudan değişiklik yapmazsınız. Çalışmalarınızı kısa vadeli branchler 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 ettğiniz ve daha az stabil ola bilen **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 halukarda nası bir branch'ing 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 sarfetmek 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 projenizi 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 branchlerinizin hepsini bu ana branch'i baz alarak oluşturup değişiklierinizi tamalayıp kalite kontrol sürecinizi (testler, kod okuma vs) de işlettikten sonra bu değişiklikeri tekrar ana branch'iniz olan master'a entegre etmeliziniz.

Diğer yandan siz kendi konu bazlı branch'inizde değişiklikleri yaparken ekip arkadaşlarınız da arada kendi değişikliklerini master branch'e entegre ediyor olacaklarıdır. 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; değişiklikerinizi kalite kontrol süreçlerinizi işletmeden ana branch'inizi 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'de 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 branchler ile yerel branchleri 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 Stratejiler

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 branch'ing 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 ağıda yer alan remote bi repository oluşturmanıza da gerek olmayacaktır.

Ancak takım çalışması söz konusu olduğunda, takımdaki geliştiricilerin birlikte çalışabilmesi için herkesin değişikliklerini ortak bir alanda yayınlaması ve diğerlerinin de bu ortak alan üzerinden bu değişiklikleri kendi branch'lerine entegre etmesi gerekecektir. Bu durumda başvuracağınız en etkin araç Git'deki Remote Repository işlevleridir. Remote repository'leri en basit anlamda tüm ekibin erişimi olan dosyal 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 yapılabiliyor. Ancak tüm bu benzerliklere rağmen remote repository'ler için Working Copy (aktif branch'deki dosyaların diskimizdeki kopyaları) yapısı geçerli değildir, remote rpository'lerde sadece Git'in veritabanının 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 repoository'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 srasında 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 Respository'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 respository 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

Yukarıdaki ekran görüntüsünde ilk komutumuz olan **git remote add** ile local repository'miz ile remote repository'miz arasındak 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.

Dikkat edecek olursanız local repository'lerimiz hala yerinde duruyor ancak listemizde ilave olarak **origin/HEAD** ve **origin/master** isimli iki remote kaydı var. Pekiyi 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 trasferi gerçekleşmez.

> **Remote Repository bilgileri güncel olmayabilir!** Git remote repository'ler ile ilgili yerel diskinizde bir takım bilgileri içerir. Ancak Git arak planda otomatik olarak bu bilgileri sizin için belirli aralıklarda güncellemez! Bu işlemin gerçekleşmesi ve sizin diğer takım arkadaşlarınız 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.

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

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

Bilgilerini güncellediğimiz git101_ornek/master isimli branch'de değişiklikler yapmak için öncelikle bu branch'i baz alarak yeni bir local branch oluşturup dosyaların Working Copy alanımıza kopyalanmasını sağlamamız gerekiyor. Bunun için **git checkout** komutunu **--track** parametresi ile kullnıyoruz.

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 versyon kontrolüne ilişkin işlemlerimizi bitiriyoruz. Son adım olarak da **git push** komutu ile localdeki bu değişikliklerimizi remote branch'de yayınlıyoruz.

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

Remote Değişiklikleri Entegre Etmek

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

Remote branch'deki değişiklikler idirmek 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 commti sırasında girilen mesaj gibi)

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

> Remote branchdeki değişikliklerin bilgilerini indirmek için kullanılan **fetch** (türkçe anlamı getirmek) 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 değişiiklik 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.

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.

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

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 varsaylım. 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.

> 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 durduğunuz göreceksiniz. Bunun nedeni **git branch -dr** komutundaki seçeneklerden **r** seçeneğinin suncudaki branch'i değil yerel bilgisayarınızda remote branch bilgilerini siler. Bu değişikliğin suncuda da geçerli olması için yani sunucudaki branch'i de silmek için **git push origin :superyeniozellik** komutu ile değişikliği bir anlamda remote repositry'de yayınlamanız gerekiyor.

> 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 alabilmemızı ağlamasıdır.

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.

Versyon 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 düzeltttiğiniz 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 almaycak.

Local Değişiklikleri Geri Almak

Henüz commit etmediğimiz değişikliklikliklere 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, **fit 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 değişkliğinizi geri almak için otomatik olarak yeni bir commit oluşturur ve geri alma işlemi bu commit sayesinde değişiklik tarihçesinde görünür hale gelir.

Yukarıdaki ekran görüntüsünde ilk önce **git revert** komutunu çalıştırdık. Bu komutun en önemli parametresi geri almak istediğimiz commit'in hash değeri (hash'in ilk altı 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 komu ise **git reset** komutun. Bu komut da herhangi bir bilginizi silmeden işlemi gerçekleştirir, ancah git revert komutundan farklı olarak otomatik yeni bir commit üretmeden değişikliğinizi geri almanızı sağlar.

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 kulllanırken 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 değiştirilebiliri
> - 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 commmitlerdeki iki versiyonunun diff'ini alıyoruz.

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 karçılaştırılan versiyonların hash değeri sonuncusu da (1000644) dosya modu bilgisidir.

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 örğrenmiştik. **git status** komutu ile dosyaların içeriğindeki değişiklikleri göremeyiz. İçerik değişikliklerini de görmek için doğrudan **git diff** komutunu herhangi bir parametre veya seçenek belirteden kullanabilirsiniz.

Sadece Staging Area'ya commit edilmek üzere eklenmiş/çıkarılmış dosyalardaki değişiklikler 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 paremetre olarak verebilirsiniz. Örneğin **master** ile **superyeniozellik** branch'ini karşılaştırmak için *git diff* komutu aşağıdaki gibi oalcaktır

> **$ 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 karşılaştırabiliriz. 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 ev çakışmaları en kolay ve efektif bir şekilde nasıl çözebileceğimizi ele alacağız.

Git ile güvendesiniz

Git ile çalışıyorsanız istediğiniz zaman yanlış yaptığınız değişiklik entegre etme işlemini 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 hiç bir zaman sunucu tarafında olamamasıdır. Böylece merge işlemi sırasında meydana gelen çakışmada takım arkadaşlarınız etkilenmeyecektir.

Çakışma Nasıl Oluşur?

Git'de **merge** işlemi bşaka 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 işlemini 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 dısyada 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.

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 karşılaşıyoruz.

- **<<<<<<<<< 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 birerbir 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 Ayırdetmek

Normal commit'leri yazılım geliştiriciler ince eleyip sık dokuyarak oluşturulurlar, diğer yandan Merge Commitler ise Git tarafından otomatik oluşturulurlar. 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 herhangib 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 kullanılabilecek diff/merge araçları
- Git servisleri
- Kaynaklar ve Referanslar

Görsel Git İstemcileri

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

Atlassian SourceTree

SourceTree ücretsiz bir uygulamadır ve Mac OS X, Windows ve Linux işletim sistemlerinde çalışmaktadır.

SourceTree'yi bu linkten indirebilirsiniz

Linux'a Özel İstemciler

- GitEye : Linux'un yanısıra OSX ve Windows işletim sistemlerinde de kullanılabilir.
- gitg
- giggle
- GitForce
- RabbitVCS

Tower

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

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.

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 üzeriden 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 sunucularınzdadır
- 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 paylaşıma 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 ücretlis hizmet seçeneği de var.

BitBucket 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 arttırabilirsinizi.

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