



Özcan Acar

Pratik GIT

Git Version Kontrol Kullanım Rehberi

PRATİK GIT

Pratik GIT

Yazılımcılar için Git versiyon kontrol kullanımı ve yazılım geliştirme rehberi

Özcan Acar

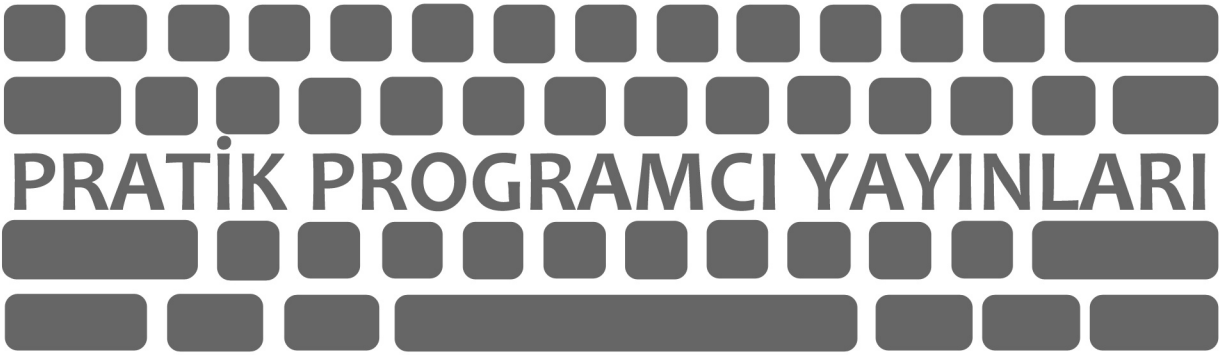


Pratik Programcı 3

PRATİK GIT . Copyright © 2016 Pratik Programcı Yayınları.

Tüm telif ve yayın hakları Pratik Programcı Yayınları'na aittir. Telif hakkı sahibinin yazılı izni olmadan kısmen ya da tamamen alıntı yapılamaz, kopya edilemez, çoğaltılamaz, dağıtılamaz ve yayınlanamaz.

Yazar: Özcan Acar
Yayınevi: Pratik Programcı Yayınları
İlk sürüm: Nisan 2016
Kapak tasarımı: Ahmet Yıldırım
Düzeltili: Ahmet Yıldırım
Satış: <http://www.pratikprogramci.com>



pratikprogramci.com

Asya'daki (kızım) Vatan'ım (eşim), Türkiye'm için ...

Bilgi Paylaşım İle Çoğalır

Lütfen bu kitabın ve ihtiva ettiği bilginin yaygınlaşması için kitabı tanıdıklarınız için paylaşınız.

Pratik Programcı Yayınları

<http://www.pratikprogramci.com>

Bölüm Başlıkları

Kitap 8 bölümden oluşmaktadır. Ana bölüm başlıkları şunlardır:

- 1.Bölüm - Version ve Git Hakkında
 - 2.Bölüm - Git Temel İşlemler
 - 3.Bölüm - Git Dal Yönetimi
 - 4.Bölüm - Remote Depo Kullanımı
 - 5.Bölüm - Git İş Akış Modelleri
 - 6.Bölüm - Git Nesne Veri Tabanı
 - 7.Bölüm - Git Pratik Bilgiler
 - 8.Bölüm - Depo Bağımlılıkları
-

İçindekiler

Bilgi Paylaşım İle Çoğalır	12
Bölüm Başlıkları	16
Önsöz	21
Yazar Hakkında	21
Neden Pratik Git?	21
Kitap Kim İçin Yazıldı?	22
Kitap Nasıl Okunmalı?	22
Yazar İle İletişim	22
PratikProgramci.com	22
1. Bölüm	25
Version ve Git Hakkında	25
Versiyon Nedir?	26
Versiyon Kontrolü Nedir?	31
Versiyon Kontrolü Nasıl Uygulanır?	32
Versiyon Kontrol Sistemleri	33
Yerel Versiyon Kontrolü	33
Merkezi Versiyon Kontrolü	33
Dağıtık Versiyon Kontrolü	34
Özet	37
2. Bölüm	39
Git Temel İşlemler	39
Git Kurulumu	40
Yerel Depo Oluşturma	42
Depoya Dosya Ekleme	43
Depodan Dosya Silme	46
Git Ignore	48
Git Config	49
Yalın (Bare) Git Depoları	51
Git Clone	52
Git Pull	56
Git Log	58
Son Commitin Değiştirilmesi	63
Git Revert ile Değişikliklerin Geri Alınması	65
Etiket Kullanımı	68
Özet	72
3. Bölüm	74
Git Dal Yönetimi	74
Branch Konsepti	75
Dalların Budanması	81
Git Fast-Forward Merge	96
Deponun Tarihçesi	99
Git Rebase	100
Git Reset	113
Git Clean	117
Git Checkout	119
Özet	127

4. Bölüm	129
Remote Depo Kullanımı	129
Git Remote	130
Merkezi Depoya Yeni Bir Dal Nasıl Eklenir?	134
Git Fetch	136
Git Pull	139
Git Push	141
Özet	143
5. Bölüm	145
Git İş Akış Modelleri	145
Git Flow Workfow	153
First Parent Geçmişi	166
Özet	168
6. Bölüm	170
Git Nesne Veri Tabanı	170
Git Nesne Veri Tabanı	177
Özet	183
7. Bölüm	185
Git Pratik Bilgiler	185
Git Stash	186
İnteraktif Staging	191
Pratik Bilgi ve İşlemler	196
Git Hooks	199
Özet	201
8. Bölüm	203
Depo Bağımlılıkları	203
Submodül Kullanımı	204
Subtree Kullanımı	212
Özet	216
BTSoru.com	218
KurumsalJava.com	219
EOF (End Of Fun)	220

Önsöz

Merhaba, ismim Özcan Acar. Git benim günlük çalışma hayatımın bir parçası. Git'i kullanmadığım gün yoktur. Bu denli hayatımın bir parçası haline gelen bir yazılım aracı hakkında bir kitap yazma fikri çok zamandır aklımdaydı. Şimdi bu kitabı elinizde tutuyorsunuz.

Bu kitabı salt okuyarak Git'i tam anlamıyla kavrayabileceğinizi düşünmüyorum. Bu yüzden sizden ricam, kitabın değişik bölümlerini okurken bilgisayar başında olun ve bir Git bash konsolu ile verdiğim örnekleri uygulayın. Bu şekilde çok kısa bir zamanda Git'e hakim olacağınızdan emin olabilirsiniz.

Bu kitabı çok büyük zevk alarak yazdım, çünkü yazarken ben de çok şey öğrendim. Umarım kitap beklentilerinizi tatmin eder.

Yazar Hakkında

1974 İzmir doğumluyum. İlk ve orta öğrenimimi İzmir'de tamamladıktan sonra Almanya'da bulunan ailemin yanına gittim. Doksanlı yılların sonunda Almanya'nın Darmstadt şehrinde bulunan FH Darmstadt üniversiteden bilgisayar mühendisi olarak mezun oldum. 2001 senesinde ilk kitabım Perl CGI, 2008 senesinde Java Tasarım Şablonları ve Yazılım Mimarileri isimli ikinci kitabım, 2009 yılında Extreme Programming isimli üçüncü kitabım Pusula tarafından yayımlanmıştır.

KurumsalJava.com, Mikrodevre.com, SmartHomeProgrammer.com ve DevOnBike.com adresleri altında blog yazıyorum. Kurduğum BTSoru.com'da bana yazılımla ilgili sorularınızı yöneltebilirsiniz.

Yazdığım kitaplara <http://www.pratikprogramci.com> adresinden ulaşabilirsiniz.

Neden Pratik Git?

Modern yazılım dendiğinde akla gelen ilk araçların başında versiyon kontrol sistemleri gelir. Versiyon kontrol araçlarının kullanımı sayesinde yüksek derecede karmaşık yazılım sistemlerinin geliştirilmeleri mümkün hale gelmiştir. Günümüzde bu araçların başında da Git versiyon kontrol sistemi gelmektedir. Git başlı başına bir başarı hikayesidir. Temeli Linux işletim sisteminin çekirdeğini geliştiren Linux Torvalds tarafından atılmıştır. Git hiçbir ticari amaç güdülmeden, sadece Linux işletim sistemi çekirdeğinin geliştirildiği yazılım projesinin gereksinimlerini tatmin

etmek için geliştirilmiştir.

Linux çekirdeği gibi bir yazılım ürünü için iyi olan bir araç diğer yazılım projeleri için kötü olamaz. Nitekim kolay kullanımı ve sunduğu ileri seviye versiyon kontrol teknikleri ile Git birçok projede kullanılır hale geldi. Git artık modern yazılım dendiğinde akla gelen ilk araç. Açık kaynaklı bir araç olması Git'in hızlı bir şekilde yazılım projelerinde yayılmasını desteklemektedir.

Programcı olarak çalıştığım projelerde Git'in kullanımını teşvik ve tavsiye ediyorum. Bu kitap bünyesinde edindiğim tecrübeleri sizinle paylaşmak istedim.

Kitap Kim İçin Yazıldı?

Bu kitap Git versiyon kontrol sistemini yakından tanımak ve kullanmak isteyen yazılımcılar için yazılmıştır. Kitap tamamen programcılara hitap edebilmek amacıyla pratik uygulamalı tarzda şekillendirilmiştir.

Kitap Nasıl Okunmalı?

Elinizdeki kitabı bir kitap olarak değil, uygulama kaynağı olarak görmenizi tavsiye ederim. Kitabı satır, satır okumak yerine, bilgisayar başında kitapda yer alan örnekleri uygulamanız daha faydalı neticeler almanızı sağlayacaktır. Pratik yapmak çok kısa zamanda istediğiniz seviyeye gelmenizi kolaylaştıracaktır. Bu konudaki bir blog yazıma [bu link](#) üzerinden ulaşabilirsiniz.

Yazar İle İletişim

Kitap ile ilgili sorularınızı acar@agilementor e-posta adresime gönderebilirsiniz. Benimle iletişim kurmadan önce lütfen [BTSoru.com](#) adresinde sorunuz hakkında araştırma yapınız. Bilgi paylaşımını geniş çaplı tutmak için okurlarımın sorularına BTSoru.com'da cevap vermeye çalışıyorum. BTSoru.com'da araştırma yaparken ya da soru sorarken soruların bu kitaba ait olduğunu görebilmek için lütfen pratik-git etiketini kullanın.

PratikProgramci.com

[PratikProgramci.com](#) kaleme aldığım ve bundan sonra almayı planladığım kitapları dijital formatta sizlerle buluşturmak istediğim yeni bir eğitim platformu. Kitaplarım yanı sıra belli, başlı yazılım konularını kapsayan görsel öğrenim (screencast) modülleri oluşturarak, beğeninize sunmak

istiyorum. Geliřmeleri <http://www.pratikprogramci.com> adresinden takip edebilirsiniz.

1. Bölüm

Version ve Git Hakkında

Versiyon Nedir?

İnsanoğlu okuma, yazmayı icat etmeden önce mağara duvarlarına resimler yaparak düşüncelerini şekillendirmeye başladı. Araştırmalara göre ilk yazının Sümer’liler tarafından İsa’dan önce 3500 civarında icat edildiği söylenmektedir. O devrin insanları yazı benzeri işaretler kullanarak, ilk dokümanları oluşturmuş ve bu dokümanları iletişim aracı olarak kullanmışlardır. Günümüzde latin alfabesinde yer alan kelimeleri kullanarak bilgisayarda dijital dokümanlar oluşturuyoruz, arkadaşlarımıza e-posta gönderiyoruz ya da elimize kağıt ve kalem alarak, mektup yazıyoruz. Bu işlemlerin sonucunda dijital olan ya da olmayan bir doküman oluşuyor.

Her doküman oluşturulduğu ilk saniyeden itibaren bir versiyon ihtiva eder. Bir versiyon, dokümanın geleceğe doğru olan yolculuğunda durak yaptığı istasyonlardan birisidir. Zaman içinde doküman değişikliğe uğrar. Her değişikliğin ardından dokümanın yeni bir versiyonu oluşur. Dokümanın herhangi iki versiyonu arasındaki fark, bu iki versiyonun oluşumu için geçen zaman diliminde, doküman üzerinde değişiklik yapılmış olmasıdır. Bir dokümanın versiyonları arşivlenmediği sürece, o doküman üzerinde yapılan değişiklikler takip edilemez.

```
// Şema 1

$ ls -l
total 0
-rw-r----- 1 oacar users Mar 25 08:52 siparis001.txt
-rw-r----- 1 oacar users Mar 25 08:52 siparis002.txt
-rw-r----- 1 oacar users Mar 25 08:52 siparis003.txt
-rw-r----- 1 oacar users Mar 25 08:52 siparis004.txt
```

Şema 1 de görüldüğü gibi yapılan siparişler siparisxxx.txt isminde bir doküman içinde tutulmaktadır. Her yeni sipariş için dokümanın yeni bir versiyonu oluşturulmuş ve böylece bu dokümanın tarihçesi arşivlenmiştir. Dokümanın ismi sipariş.txt olup, 001, 002, 003 ve 004 versiyon numaralarıdır. siparis004.txt en son versiyondur ve bu doküman bünyesinde meydana gelen tüm değişiklikleri ihtiva etmektedir. Dokümanın değişik versiyonları arşivlendiği için değişik versiyonlar arasında kıyaslama (compare) yapılarak, meydana gelen değişiklikler takip (track) edilebilir.

Görüldüğü gibi bir dokümanı arşivleyebilmek için tekil olan bir versiyon numarasına ve doküman ismine ihtiyaç duymaktayız. Doküman ismi değişmezken, dokümanın yeni türevlerini oluşturduğumuzda, versiyon numarasını da değiştirmemiz gerekmektedir. Sadece bu şekilde bir dokümanın değişik versiyonlarını birbirinden ayırt edebiliriz.

Şimdi şema 1 de yer alan yapının nasıl oluştuğunu bir örnek üzerinde yakından inceleyelim.

Firmada siparişlerin arşivlenmesinden sorumlu olduğumuz için bize aşağıdaki sipariş listesinin gönderildiğini düşünelim.

```
// Şema 2

Sipariş no: 1000
Müşteri no: 55
-----
Sipariş no: 1001
Müşteri no: 197
-----
Sipariş no: 1002
Müşteri no: 100
```

Bize gönderilen listede 3 adet sipariş bulunmaktadır. Bu siparişleri arşivlemek için aşağıdaki şekilde sipariş.txt dosyasını oluşturuyor ve bu dosyaya şema 2 de yer alan içeriği ekliyoruz.

```
$ ls -l
total 0
-rw-r----- 1 oacar users Mar 25 07:52 sipariş.txt
```

Gün içinde bize yeni sipariş listesi gönderiliyor. Bu listeyi şema 3 de görmekteyiz.

```
// Şema 3

Sipariş no: 1003
Müşteri no: 50
-----
Sipariş no: 1004
Müşteri no: 197

İptal edilen siparişler
+++++
Sipariş no: 1002
Müşteri no: 100
```

Bu liste hem yeni siparişleri hem de iptal edilen siparişleri ihtiva etmektedir. Bu durumda sipariş.txt dosyasının yeni içeriği şu şekilde olacaktır:

```
// Şema 4

Sipariş no: 1000
Müşteri no: 55
-----
```

```

Sipariş no: 1001
Müşteri no: 197
-----
Sipariş no: 1002
Müşteri no: 100
-----
Sipariş no: 1003
Müşteri no: 50
-----
Sipariş no: 1004
Müşteri no: 197

İptal edilen siparişler
+++++
Sipariş no: 1002
Müşteri no: 100

```

Bu değişikliğin ardından dizin yapısı aşağıdaki gibidir:

```

// Şema 5

$ ls -l
total 0
-rw-r----- 1 oacar users Mar 25 07:52 sipariş.txt

```

Siparişleri arşivlemek için sipariş.txt isimli dosyayı kullanmaktayız. Bu dosyanın içeriğini yeni siparişleri ekleyerek değiştirmiş olmamıza rağmen, dosyanın ihtiva ettiği dokümanın yeni bir versiyonunu oluşturmadık. Yaptığımız değişiklik neticesinde yeni versiyon dosyanın içeriğinde gizli kaldı, çünkü sipariş.txt isimli dosyaya baktığımızda, dokümanın hangi versiyonda olduğunu görememekteyiz.

Zaman içinde yapılan yeni eklentiler ile sipariş.txt bünyesinde yer alan verilerin nasıl değiştiğini takip etmemiz de mümkün olmayacaktır. Dosya üzerinde yapılan her değişikliği takip edebilmek için sipariş.txt dosyasının yeni versiyonlarını oluşturmamız gerekmektedir.

Dosyanın yeni bir versiyonunu oluşturduğumuzu ifade etmek için dosya ismi yanı sıra bir versiyon numarası kullanmamız gerekmektedir. Versiyon numarasını rakamları ya da harfleri kullanarak, oluşturabiliriz. Örneğin:

```

// Şema 6

$ ls -l
total 0

```

```
-rw-r----- 1 oacar users Mar 25 07:52 siparis_A.txt
-rw-r----- 1 oacar users Mar 25 07:52 siparis_B.txt
```

ya da;

```
// Şema 7

$ ls -l
total 0
-rw-r----- 1 oacar users Mar 25 07:52 siparis_1.txt
-rw-r----- 1 oacar users Mar 25 07:52 siparis_2.txt
```

Şema 2 sipariş.txt dosyasının ilk versiyonunu, şema 4 ise ikinci versiyonunu ihtiva etmektedir. İkinci versiyondan itibaren her yeni eklenti ile yeni bir versiyon oluşturarak, dosya üzerinde yapılan değişiklikleri takip edebiliriz. Bu amaçla sipariş.txt dosyasını siparis001.txt olarak değiştirmemiz ve şema 3 de yer alan değişiklikleri yapmadan önce siparis001.txt dosyasını siparis002.txt olarak yeni bir dosyaya kopyalamamız gerekmektedir. Bu yeni yapı şema 8 de yer almaktadır.

```
// Şema 8

$ ls -l
total 0
-rw-r----- 1 oacar users Mar 25 07:52 siparis001.txt
-rw-r----- 1 oacar users Mar 25 07:52 siparis002.txt

// siparis001.txt içeriği

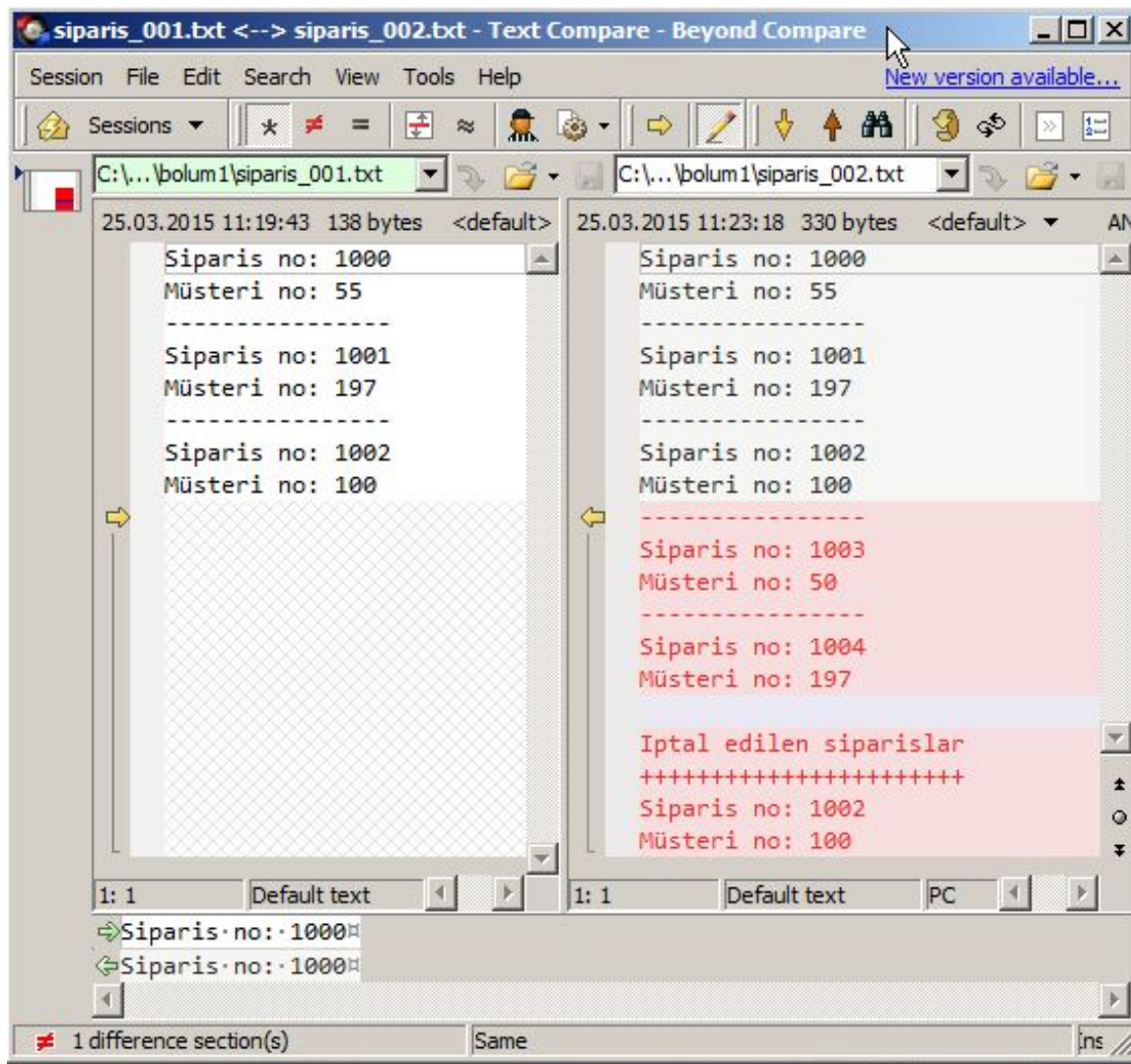
Sipariş no: 1000
Müşteri no: 55
-----
Sipariş no: 1001
Müşteri no: 197
-----
Sipariş no: 1002
Müşteri no: 100

// siparis002.txt içeriği

Sipariş no: 1000
Müşteri no: 55
-----
Sipariş no: 1001
Müşteri no: 197
```

```
-----  
Sipariş no: 1002  
Müşteri no: 100  
-----  
Sipariş no: 1003  
Müşteri no: 50  
-----  
Sipariş no: 1004  
Müşteri no: 197  
  
İptal edilen siparişler  
+++++  
Sipariş no: 1002  
Müşteri no: 100
```

Bir dosyanın yeni versiyonlarını oluşturduğumuz andan itibaren değişik versiyonları birbirleriyle kıyaslayarak, dosya üzerinde yapılan değişiklikleri takip edebiliriz. Resim 1 de [Beyond Compara](#) programı ile yaptığım siparis001.txt ve siparis002.txt dosyalarının içerik kıyaslamasını görmektesiniz.



Resim 1

Kıyaslama aracı iki dosya arasındaki farklılığı kırmızı alan içinde göstermektedir.

Versiyon Kontrolü Nedir?

Şema 8 de yer alan yapı ilkel bir versiyon kontrol sistemidir. Oluşturduğumuz bu versiyon kontrol sisteminin özelliklerini şu şekilde sıralayabiliriz:

- Her yeni bir versiyon için dosyanın yeni bir versiyon numarası ile kopyalanması gerekmektedir. Bu elden yapılması gereken bir işlem olduğundan, yeni versiyon oluştururken hata yapma potansiyeli yüksektir.
- Versiyon numaraları dosya isminde tutulmaktadır. Bu hızlı bir şekilde versiyon arama çalışmalarını zorlaştırabilir.
- Her yeni bir versiyon için yeni bir dosyanın oluşturulması gerekmektedir. Bu zaman içinde

dosya enflasyonuna neden olabilir.

- Kimin hangi versiyonu oluşturduğunu ve hangi değişiklikleri yaptığını takip etmek hemen hemen mümkün değildir.

Şema 8 de yer alan yapının ufak çaplı versiyonlama aktiviteleri için yeterli olduğunu söyleyebiliriz. Lakin çok geniş bir kod tabanına sahip bir yazılım projesinde bu tür bir versiyon kontrol sisteminin kullanılması düşünülemez. Neye ihtiyaç olduğunu belirtmeden önce, versiyon kontrol teriminin tanımını yapmamız faydalı olacaktır.

Bir dokümanın oluşum sürecinin ve değişik versiyonlarının takibi ve arşivlenmesi için kullanılan metot ve sistemlere versiyon kontrolü adı verilir. Genelde yazılım sektöründe projelerin yönetimi için versiyon kontrol sistemleri kullanılır. Birden fazla programcının kod paylaşımı ve yapılan değişikliklerin takibi için bir versiyon kontrol sisteminin kullanımı kaçınılmazdır.

Daha sonra yakından inceleyeceğimiz gibi, hazırlanan bir yazılım ürününün (program) değişik versiyonlarının oluşturulması ve bu versiyonlardaki hataların (bug) giderilmesi için kullanılan versiyon kontrol sistemi değişik araçlar ve yöntemler ihtiva etmektedir. Bu metotlar kullanılarak yazılım süreci desteklenir.

Versiyon Kontrolü Nasıl Uygulanır?

Bir versiyon kontrol sistemini depo gibi düşünebiliriz. Oluşturulan her doküman versiyon kontrol sisteminin sahip olduğu araçlar aracılığı ile bu depoya eklenir. Depo, dokümanların ve değişik versiyonlarının nasıl yönetileceğini bilir. Programcı olarak deponun içinde neler olup, bittiğini bilmek zorunda değiliz. Sadece bir dokümana ihtiyaç duyduğumuz zaman depoya başvurarak, gerekli dokümanı ediniriz. Bu doküman üzerinde değişiklik yapabilir ve değiştirilen dokümanı tekrar depoya gönderebiliriz. Bu esnada dokümanın değişik versiyonları oluşur. Oluşan değişik versiyonlar üzerinde de kafa yormamıza gerek yoktur, çünkü depo bu versiyonların yönetimi üstlenir. Depoya danışarak, bir dokümanın tarihçesini edinebiliriz. Depo hangi dokümanın kim tarafından ne zaman değiştirildiğini her zaman bilir. Kayıtlarda iz bırakmadan bir doküman üzerinde değişiklik yapmak mümkün değildir.

Versiyon kontrol sistemlerinde kullanılan depolara repository ismi verilir. Bir repository programcının kendi bilgisayarında ya da bir sunucu üzerinde yer alabilir. Programcı çalışmak üzere depodan dosya edinebilir. Bu işleme checkout ismi verilmektedir. Programcı depodan edindiği dosyalar üzerinde değişiklik yaptığı takdirde, bu dosyaların yeni versiyonlarını oluşturmuş olur. Programcının yaptığı bu değişiklikleri diğer programcılarla paylaşabilmesi için değişikliğe uğrayan dosyaların tekrar depoya gönderilmesi gerekmektedir. Bu işleme commit ya da checkin ismi

verilmektedir. Yapılan her commit ile değiştirilen dosyalar için version kontrol sistemi tarafından yeni bir revizyon oluşturulur. Revizyon numarası ile dosyanın istenilen haline erişim mümkündür. Her revizyon dosyanın o revizyonda ugradığı değişiklikleri ihtiva eder. Revizyonları kıyaslayarak, iki değişik versiyon arasındaki farkı bulmak mümkündür.

Versiyon Kontrol Sistemleri

Yazılım projerinde kullanılmak üzere birçok versiyon kontrol sistemine rastlamak mümkün. Bunlardan bazıları:

- Subversion
- Git
- CVS
- Mercurial
- ClearCase
- Team Foundation
- Visual SourceSafe

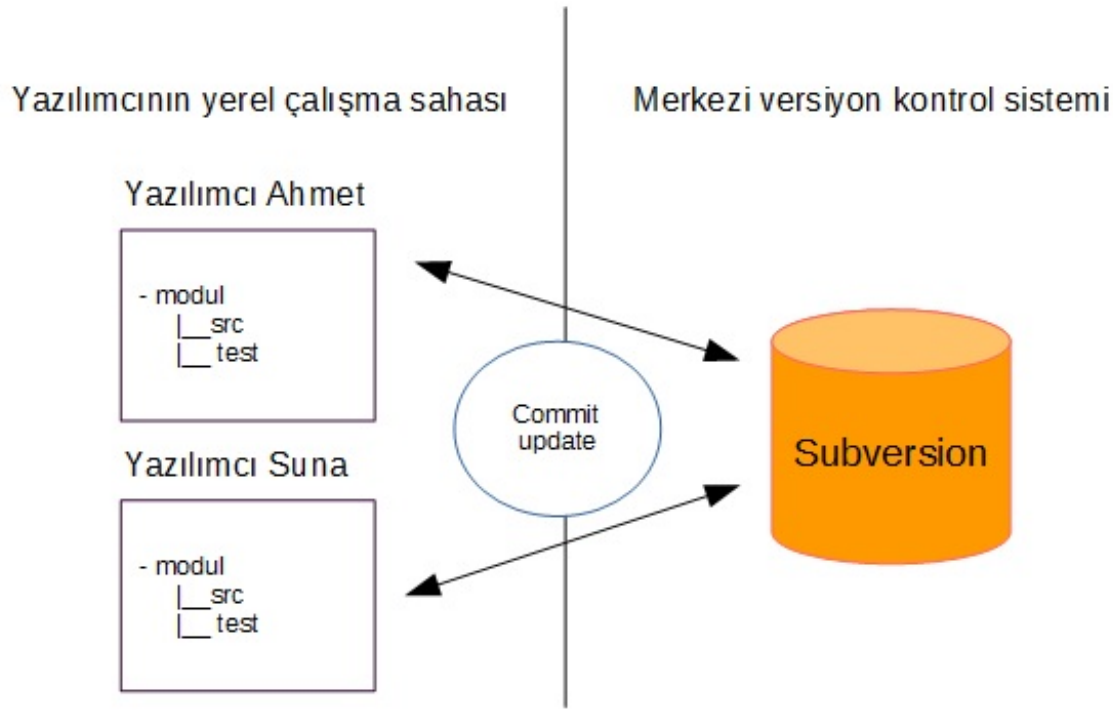
Bu ürünlerin bir kısmı açık kaynaklı olup, yazılım projelerinde lisans bedeli olmadan kullanılabilirler. Günümüzde en popüler açık kaynaklı versiyon kontrol sistemleri Subversion ve Git'dir. Versiyon kontrol sistemlerinde yerel (local), merkezi (central) ve dağıtık (distributed) mimari modelleri kullanılmaktadır.

Yerel Versiyon Kontrolü

Bunun bir örneğini şema 1 de görmekteyiz. Bu tür versiyon kontrol sistemlerinde tek bir dosyanın değişik versiyonları oluşturulur. Microsoft Office gibi ürünlerde dokümanın değişik versiyonları aynı dosya içinde tutularak, yerel versiyon kontrolü uygulanır.

Merkezi Versiyon Kontrolü

Bu tür sistemlerde versiyon kontrol sistemi tarafından oluşturulan depo bir sunucu üzerinde yer alır. Checkout ve commit işlemleri için bu sunucu ile bağlantı kurulması gerekmektedir. Örneğin Subversion bu mimariye göre çalışan bir versiyon kontrol sistemidir.



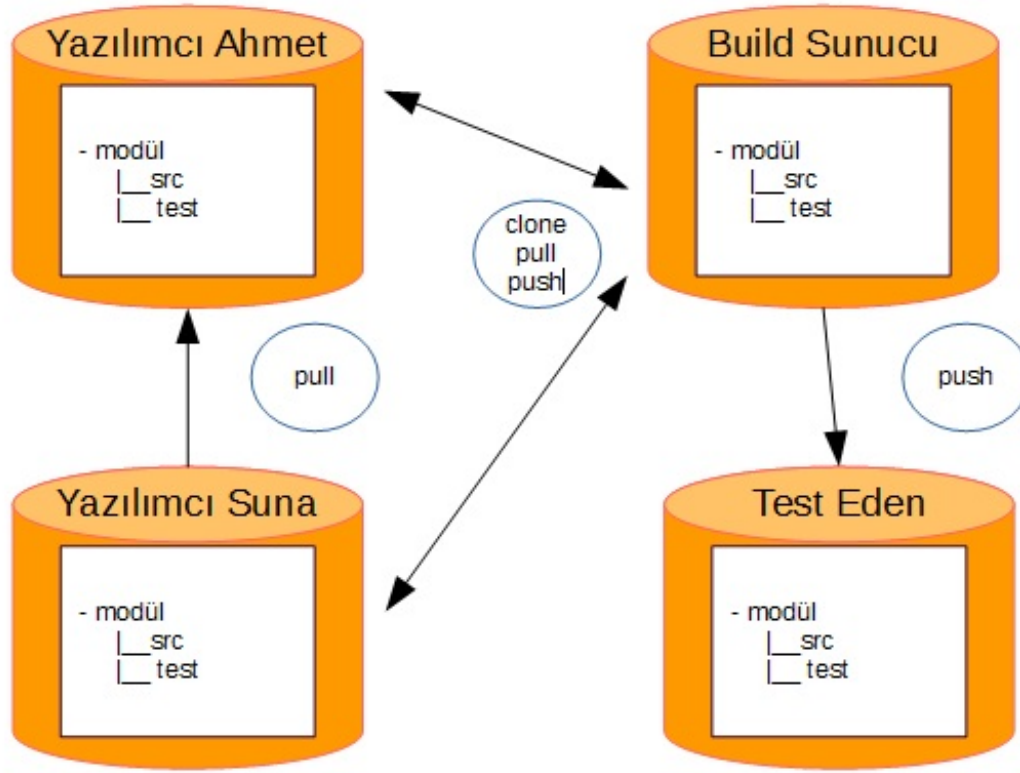
Resim 2

Kullanıcı merkezi depo üzerinde yer alan dosyaların kopyalarını checkout işlemi ile kendi bilgisayarına alır. Dosyalar üzerinde yapılan değişiklikler commit işlemi esnasında ağ üzerinden merkezi versiyon kontrol sistemine aktarılır. Sunucu ağ üzerinden erişilir olmadığı durumlarda, commit işlemini gerçekleştirmek ve yapılan değişiklikleri diğer kullanıcılar ile paylaşmak mümkün değildir.

Dağıtık Versiyon Kontrolü

Dağıtık versiyon kontrol sistemlerinde merkezi bir depo kullanımı yanı sıra yerel depo kullanımı da mümkündür. Kullanıcı hiçbir sunucuya bağımlı olmadan kendi yerel deposunu oluşturabilir ve bu depo üzerinde checkout ve commit işlemlerini gerçekleştirebilir.

Dağıtık versiyon kontrol sistemi



Resim 3

Aynı şekilde merkezi bir sunucuda bulunan bir depoyu klonlayarak, kullanmak mümkündür. Burada klonlama terimi deponun bir kopyasını oluşturma anlamında kullanılmaktadır. Kullanıcı mevcut bir depodan yola çıkarak, kendi deposunu oluşturduktan (klonladıktan) sonra, tüm checkout ve commit işlemlerini klonlanan depo üzerinde gerçekleştirir. Klon depo oluşturulduktan sonra, merkezi depo ve yerel depo birbirlerinden bağımsız hale gelirler. Kullanıcı istediği değişiklikleri kendi yerel deposunu yaptıktan sonra, bu değişiklikleri merkezi depoya aktarabilir. Bu işlemi Git terminolojisinde push ismi verilmektedir. Aynı şekilde merkezi depoda meydana gelen değişiklikleri klon olan yerel depoya aktarmak için pull işlemi gerçekleştirilir.

Resim 3 de bir dağıtık versiyon kontrol sistemi ile çalışma tarzını görmekteyiz. Yazılımcı Ahmet build sunucusunda bulunan depoyu klonlamış ve yerel bir depo oluşturmuştur. Aynı işlemi Suna'da gerçekleştirmiştir. Ahmet ve Suna yerel depoda yaptıkları her değişikliği push komutu ile sunucuya aktarmakta ve pull komutu ile sunucu üzerinden birbirlerinin değişikliklerini edinmektedirler. Bunun yanı sıra Suna pull komutu ile Ahmet'in yerel deposundan dosya değişikliklerini edinebilmektedir. Build sunucusu push komutu ile her yapılandırma (build) işleminden sonra kodu test edilmek amacıyla başka bir depoya aktarmaktadır.

Özet

- Her doküman oluşturulduğu ilk saniyeden itibaren bir versiyon ihtiva eder. Bir versiyon, dokümanın geleceğe doğru olan yolculuğunda durak yaptığı istasyonlardan birisidir. Dosyanın yeni bir versiyonunu oluşturduğumuzu ifade etmek için dosya ismi yanı sıra bir versiyon numarası kullanmamız gerekmektedir.
- Bir dokümanın oluşum sürecinin ve değişik versiyonlarının takibi ve arşivlenmesi için kullanılan metot ve sistemlere versiyon kontrolü adı verilir.
- Bir versiyon kontrol sistemini depo gibi düşünebiliriz. Versiyon kontrol sistemlerinde kullanılan depolara repository ismi verilir.
- Versiyon kontrol sistemleri yerel, merkezi ve dağıtık olmak üzere üç gruba ayrılmaktadırlar.

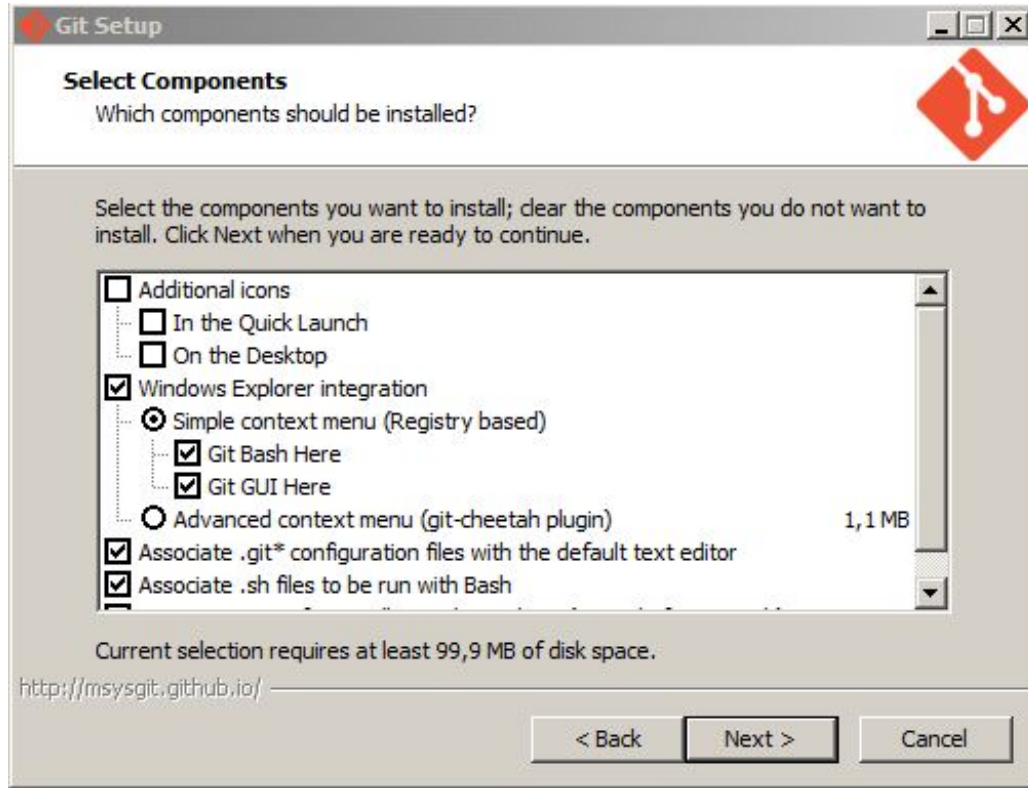
2. Bölüm

Git Temel İşlemler

Git Kurulumu

<http://git-scm.com/downloads> adresinden Git'in güncel sürümünü edinebilirsiniz. Benim bu kitapta yer alan örnekler için kullandığım Git sürümü 1.9.5 dir.

Eğer Windows işletim sistemini kullanıyorsanız, kurulum esnasında Git Bash opsiyonu seçmenizi tavsiye ederim. Git Bash isminde de anlaşıldığı gibi bir Unix Bash konsoludur ve Git komutlarının konsol üzerinde kullanımını mümkün kılmaktadır.



Resim 1

Kurulum neticesini kullandığınız işletim sisteminin konsol programı aracılığı ile kontrol edebilirsiniz.

// Şema 1

```
[oacar@lin ~]$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        <command> [<args>]
```

The most commonly used git commands are:

add	Add file contents to the index
bisect	Find by binary search the change that introduced a bug
branch	List, create, or delete branches
checkout	Checkout a branch or paths to the working tree
clone	Clone a repository into a new directory
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
fetch	Download objects and refs from another repository
grep	Print lines matching a pattern
init	Create an empty Git repository or reinitialize an existing one
log	Show commit logs
merge	Join two or more development histories together
mv	Move or rename a file, a directory, or a symlink
pull	Fetch from and integrate with another repository or a local branch
push	Update remote refs along with associated objects
rebase	Forward-port local commits to the updated upstream head
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index
show	Show various types of objects
status	Show the working tree status
tag	Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some concept guides. See 'git help <command>' or 'git help <concept>' to read about a specific subcommand or concept.

[oacar@lin ~]\$

Eğer git komutunu girdiğinizde hata alıyorsanız, bu taktirde Git'in kurulumunun yapıldığı bin dizinini çalıştırılabilir program dizinleri listesine almanız gerekmektedir. Bunu su şekilde yapabilirsiniz:

```
// Şema 2

Windows işletim sistemleri için:
_____

c:\>set PATH=c:\git\bin;%PATH%

Linux işletim sistemi için:
_____

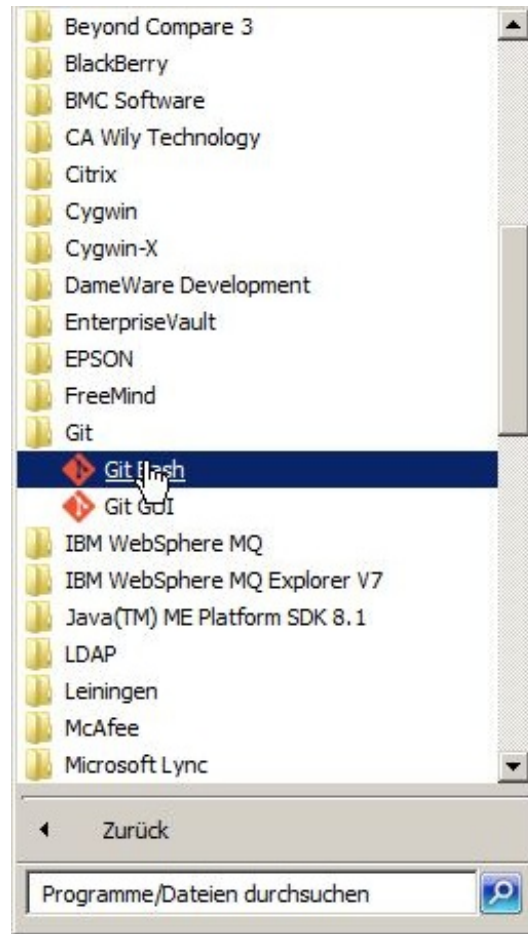
[oacar@lin ~]export PATH=/opt/git/bin:$PATH
```

Kullandığınız Git sürüm numarasına şu şekilde ulaşabilirsiniz:

```
[oacar@lin ~]$ git --version
git version 1.9.5
[oacar@lin ~]$
```

Görüldüğü gibi kitapta yer alan Git örneklerini Linux tabanlı bir bilgisayarda oluşturdum. Örnekleri hem Windows konsol hem Windows Git Bash hem de Linux ya da MacOS tabanlı bir sistemde aynı şekilde uygulayabilirsiniz.

Windows işletim sisteminde çalışıyorsanız, Git Bash programı ile Unix vari bir ortamda Git ile çalışabilirsiniz. Git Bash programını Git kurulum dizininde bulabilirsiniz.



Resim 2

Yerel Depo Oluşturma

Git ile versiyon kontrolü yapabilmek için öncelikle yerel bir deponun (repository) oluşturulması gerekmektedir. Yerel bir Git deposunu şema 3 de yer aldığı şekilde oluşturabiliriz.

```
// Şema 3

[oacar@lin workspace]$ git init
Initialized empty Git repository in /home/oacar/workspace/.git/
[oacar@lin workspace]$
```

Bu örnekte yerel depoyu /home/oacar/workspace dizininde oluşturmuş olduk. Dizinin içeriğine baktığımızda, dizinin boş olduğunu görmekteyiz:

```
// Şema 4

[oacar@lin workspace]$ ls -l
total 0
```

Bir şeyler görebilmek için daha yakından bakmamız gerekiyor:

```
// Şema 5

[oacar@lin workspace]$ ls -la
total 12
drwxr-x--- 3 oacar users  4096 Mar 26 15:21 .
drwx----- 3 oacar appsupp 4096 Mar 26 15:15 ..
drwxr-x--- 7 oacar users  4096 Mar 26 15:21 .git
```

Yakından baktığımızda, workspace dizini içinde .git isimli bir dizinin olduğu görmekteyiz. Bu oluşturduğumuz yerel Git deposunun yer aldığı dizindir. Git depo üzerinde yapılan tüm işlemleri bu dizin içinde yer alan dosyalar aracılığı ile takip eder. Bu dizini Git'in veri tabanı olarak düşünebiliriz. Bu dizinde yer alan yapıyı kitabın ilerleyen bölümlerinde yakından inceleyeceğiz.

Depoya Dosya Ekleme

Şema 5 de görüldüğü gibi boş bir Git deposuna sahibiz. Şimdi bu depoya yeni dosyaların nasıl eklendiğini bir örnek üzerinde inceleyelim.

```
// Şema 6

[oacar@lin workspace]$ echo "Bu bir test..." > dosya1.txt

[oacar@lin workspace]$ echo "Baska bir test..." > dosya2.txt

[oacar@lin workspace]$ ls -l
```

```
total 8
-rw-r----- 1 oacar users 15 Mar 26 15:36 dosya1.txt
-rw-r----- 1 oacar users 18 Mar 26 15:36 dosya2.txt
```

Workspace isimli dizinde iki dosya oluşturduk. Bu dosyaların içeriği şema 6 da yer almaktadır. Deponun son durumunu kontrol etmek için status komutunu kullanabiliriz:

```
// Şema 7

[oacar@lin workspace]$ git status
On branch master

Initial commit

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

    dosya1.txt
    dosya2.txt

nothing added to commit but untracked files present (use "git add" to track)
[oacar@lin workspace]$
```

Untracked files bölümünde, henüz depoya eklenmemiş dosyaları görmekteyiz. Bu dosyaları aşağıdaki şekilde depoya ekleyebiliriz:

```
// Şema 8

[oacar@lin workspace]$ git add --all

[oacar@lin workspace]$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   dosya1.txt
    new file:   dosya2.txt
```

Add komutu ile değişikliğe uğrayan dosyaların isimlerini kullanarak ya da --all ile mevcut tüm dosyaları depoya ekleyebiliriz. Ama git add ile gerçekten dosyaları depoya eklemiş olduk mu? Add işleminden sonra git status komutunu girdiğimizde, dosyalarımızın "Changes to be committed"

bölümünde yer aldığını görmekteyiz. "Changes to be committed" depoya eklenmeyi bekleyen değişiklikler anlamına gelmektedir. Demek oluyor ki git add ile dosyaları depoya doğrudan eklemiş olmuyoruz. Add komutu depoya eklenmesi gereken dosyaların tayin edilmesinde kullanılan commit öncesi komuttur. Add ile depoya ekleme istediğimiz dosyaları seçer ve commit komutu ile seçilen bu dosyaları depoya ekleriz. Bu sebepten dolayı git add ile tanımladığımız dosyalar "Changes to be committed" (şema 8) bölümünde yer almaktadır.

Git terminolojisinde commit öncesi add komutu ile dosyaları eklendiği alana index ya da staging area ismi verilmektedir. Add komutunu şu şekilde kullanabiliriz:

```
// Şema 9

git add <dosya>      ; Bir dosyayı depoya eklemek için seçer.

git add <dizin>      ; Tanımlanan dizin içindeki tüm dosyaları
                    depoya eklemek için seçer.

git add -p           ; İnteraktif bir şekilde depoya eklenecek
                    dosya içeriklerinin seçilmesini sağlar.
```

Add işleminde sonra commit komutu ile mevcut dosyaları şimdi depoya ekleyebiliriz:

```
// Şema 10

[oacar@lin workspace]$ git commit -m "initial commit"
[master (root-commit) 96985f5] initial commit
Committer: Linux User <oacar@lin>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

2 files changed, 2 insertions(+)
create mode 100644 dosya1.txt
create mode 100644 dosya2.txt
```

Dosyaların depoya eklenebilmesi için gerekli commit komutunun kullanımını şema 10 da

görmekteyiz. Burada -m parametresi commit işlemi için bir yorum yazmamızı mümkün kılmaktadır. Bu parametrenin kullanımı zorunludur yani açıklama yapmadan depoya dosya eklemek mümkün değildir. Eğer -m kullanılmazsa, Git şema 11 de görüldüğü gibi açıklama girilmesini sağlamak amacıyla commit mesajını bir editörde açar.

```
// Şema 11

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Linux User <oacar@lin>
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   dosya1.txt
#       new file:   dosya2.txt
```

Dosyaların depoya eklendiğinden emin olmak için git status komutunu kullanabiliriz:

```
// Şema 12

[oacar@lin workspace]$ git status
On branch master
nothing to commit, working directory clean
```

Şema 12 de görüldüğü gibi her iki dosyayı yeni oluşturduğumuz depoya eklemiş olduk.

Depodan Dosya Silme

Depoda bulunan bir dosyayı git rm komutu ile silebiliriz. Şöyle ki:

```
// Şema 12.1

[oacar@lin workspace]$ git status
On branch master
nothing to commit, working directory clean

[oacar@lin workspace]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 20 11:39 dosya1.txt
```

```
-rw-r----- 1 oacar users 0 Apr 20 11:39 dosya2.txt

[oacar@lin workspace]$ rm dosya1.txt

[oacar@lin workspace]$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    dosya1.txt

no changes added to commit (use "git add" and/or "git commit -a")

[oacar@lin workspace]$ git rm dosya1.txt
rm 'dosya1.txt'

[oacar@lin workspace]$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    dosya1.txt

[oacar@lin workspace]$ git commit -m "dosya1.txt silindi"
[master 2c9e304] dosya1.txt silindi
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 dosya1.txt

[oacar@lin workspace]$ git status
On branch master
nothing to commit, working directory clean
[oacar@lin workspace]$
```

Silmek istediğimiz dosyayı `rm dosya1.txt` ile sildikten sonra, deponun son durumunu `git status` ile kontrol ediyoruz. Görüldüğü gibi aşağıdaki çıktıyı aldık:

```
// Şema 12.2

[oacar@lin workspace]$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    dosya1.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Fiziksel olarak sildiğimiz dosya dosya1.txt depo bünyesinde silindi olarak işaretlenmiş durumda. Bu dosyayı depodan da silmek için `git rm dosya1.txt` komutunu kullanmamız gerekiyor. Bu işlemin ardından dosya1.txt depodan silinmeye hazır hale gelmiştir. Dosyayı depodan silmek için `git commit` komutunu kullanıyoruz. Bu şekilde dosya1.txt dosyasını hem fiziksel olarak bilgisayarımızdan hem de Git deposundan silmiş olduk.

Git Ignore

Bazen çalışma esnasında Git deposuna eklenmemesini istediğimiz dosyalar oluşabilir. Örneğin bir uygulama derlendikten sonra oluşan .dll ya da .class ekli dosyaların depoya eklenmemesi gerekir, çünkü bu tür dosyalar yapılandırma (build) işlemleri esnasında sürekli yeniden oluşturulurlar. Bu tür dosyaları göz artı etmek için aşağıda görüldüğü gibi .gitignore ismini taşıyan bir dosya oluşturmamız ve bu dosyada depoya eklenmemesini istediğimiz ve git işlemlerinde göz ardı edilecek dosya isimlerini tutabiliriz.

```
// Şema 12.3

[oacar@lin workspace]$ echo "*.test" > .gitignore

[oacar@lin workspace]$ ls -la
total 16
drwxr-x--- 3 oacar users 4096 Apr 20 13:39 .
drwx----- 5 oacar appsupp 4096 Apr 20 13:39 ..
-rw-r----- 1 oacar users 0 Apr 20 11:39 dosya2.txt
drwxr-x--- 8 oacar users 4096 Apr 20 13:39 .git
-rw-r----- 1 oacar users 7 Apr 20 13:39 .gitignore

[oacar@lin workspace]$ touch test.123

[oacar@lin workspace]$ touch abc.test

[oacar@lin workspace]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 20 13:40 abc.test
-rw-r----- 1 oacar users 0 Apr 20 11:39 dosya2.txt
-rw-r----- 1 oacar users 0 Apr 20 13:46 test.123

[oacar@lin workspace]$ git status
On branch master
```



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

        .gitignore
        test.123

nothing added to commit but untracked files present (use "git add" to track)
```

Echo komutu ile .gitignore dosyasını oluşturduk ve ilk satırını *.test olarak tanımladık. Bu dosyanın içeriği şimdi şu şekildedir:

```
[oacar@lin workspace]$ cat .gitignore
*.test
[oacar@lin workspace]$
```

*.test ile eki test olan tüm dosyaları göz ardı etmek istediğimizi tanımlamış olduk. Akabinde test.123 ve abc.test isimlerinde iki yeni dosya oluşturuyoruz. Aşağıda görüldüğü gibi dizin yapımız şimdi şu şekildedir:

```
[oacar@lin workspace]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 20 13:40 abc.test
-rw-r----- 1 oacar users 0 Apr 20 11:39 dosya2.txt
-rw-r----- 1 oacar users 0 Apr 20 13:46 test.123
```

Git status komutunu girdiğimizde, aşağıdaki çıktıyı alıyoruz:

```
[oacar@lin workspace]$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        test.123

nothing added to commit but untracked files present (use "git add" to track)
```

Görüldüğü gibi Git abc.test dosyasını göz ardı etmiştir ve sadece .gitignore ve test.123 dosyalarının değişikliğe uğradığını ve depoya eklenebileceğini bildirmektedir. Göz ardı ayarlarının kalıcı olmasını sağlamak için .gitignore dosyasını depoya ekleyebiliriz.

Git Config

Git config komutu ile Git kurulumu ve depolar için gerekli genel ve kullanıcı bazlı ayarlar yapmak mümkündür.

Şema 10 da yer alan içeriğe baktığımızda, commit esnasında kullanılacak kullanıcı bilgilerinin git config ile oluşturulabileceğini görmekteyiz. Commit işleminde kullanılacak şekilde depo genelinde geçerli kullanıcı ismi ve e-posta adresini şu şekilde tanımlayabiliriz:

```
// Şema 14

[oacar@lin workspace]$ git config --global user.name Özcan
[oacar@lin workspace]$ git config --global user.email acar@agilementor.com
```

Örneğin Git işlemleri esnasında kullanmak istediğimiz editörü şu şekilde tanımlayabiliriz:

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

Git komutları için alias isimler şu şekilde oluşturabiliriz:

```
git config --global alias.st status
```

Bu tanımlamanın ardından git st yazarak, git status işlevini elde edebiliriz.

Git konfigürasyon ayarlarını üç değişik dosyada tutmaktadır. Bunlar:

- **<depo-dizini>/.git/config** - depoya has genel ayarları ihtiva eder.
- **<kullanıcı-dizini>/.gitconfig** - Kullanıcıya has ayarlar, örneğin kullanıcı ismi ya da e-posta adresi.
- **/etc/gitconfig** - Sistem genelinde geçerli Git ayarlarını ihtiva eder.

Bu dosyalar arasında çakışma olduğunda, kişisel ayarlar depo ayarlarını, depo ayarları da sistem ayarlarını ezmektedir.

Şema 14 de yer alan işlemleri yaptıktan sonra, kullanıcı ayarlarının yer aldığı dosyanın içeriği şu şekilde olacaktır:

```
// Şema 15

[oacar@lin ~]$ pwd
/home/oacar
[oacar@lin ~]$ cat .gitconfig
[user]
    name = Özcan
```

```
email = acar@agilementor.com
[ocar@lin ~]$
```

Depo ayarlarına aşağıdaki şekilde erişebiliriz:

```
// Şema 16

[ocar@lin .git]$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[ocar@lin .git]$
```

Yalın (Bare) Git Depoları

Şimdi üzerinde çalıştığımız Git deposuna tekrar bir göz atalım.

```
// Şema 17

[ocar@lin my_workspace]$ ls -la
total 20
drwxr-x--- 3 ocar users 4096 Apr  7 11:07 .
drwx----- 4 ocar appsupp 4096 Apr  7 11:07 ..
-rw-r----- 1 ocar users 15 Apr  7 10:58 dosya1.txt
-rw-r----- 1 ocar users 18 Apr  7 10:58 dosya2.txt
drwxr-x--- 8 ocar users 4096 Apr  7 10:58 .git
[ocar@lin my_workspace]$
```

Şema 5 de oluşturduğumuz Git deposu şema 17 de görülen .git dizini içinde yer almaktadır. Peki dosya1.txt ve dosya2.txt nedir? Bu dosyalar normal şartlar altında deponun içinde yer alan ve versiyon kontrol sisteminin kontrolündeki dosyalardır. Git kendi kontrolündeki dosyalar üzerinde yapılan işlemleri ve dosyaları .git dizini üzerinden yönetir. Şema 17 de hem deponun kendisini hem de depo aracılığı ile kullandığımız dosyaları görmekteyiz. Git terminolojinde deponun kontrolünde olan ve üzerinde çalışılan dosyalara working copy ya da working tree ismi verilmektedir. Bir depoyu klonladığımız andan itibaren depoda yer alan tüm dosyaların bir kopyasını edinmiş oluruz ve böylece bir working copy üzerinde çalışırız. Şema 17 de hem Git deposunu hem de working copy yapısını görmekteyiz. Bu depo üzerinde aktif olarak çalıştığımızı göstermektedir.

Merkezi bir Git deposu oluştururken, üzerinde çalışılan dosyaların yani working copy nin deponun

bir parçası olması tercih edilen bir yöntem değildir. Daha ziyade Git deposunun sadece .git dizininden oluşacak şekilde yalın olması ve klonlama işlemi ile working copy oluşturulması tercih edilen çözümdür.

Working copy ihtiva etmeyen yalın bir Git deposunu şu şekilde oluşturabiliriz:

```
// Şema 18

[oacar@lin ~]$ mkdir repo
[oacar@lin ~]$ cd repo/
[oacar@lin repo]$ git init --bare projem.git
Initialized empty Git repository in /home/oacar/repo/projem.git/
[oacar@lin repo]$ ll
total 4
drwxr-x--- 7 oacar users 4096 Apr  7 11:43 projem.git
[oacar@lin repo]$
```

Bu örnekte projem.git ismini taşıyan yeni bir depo oluşturduk. Deponun içeriği şema 19 da yer almaktadır:

```
// Şema 19

[oacar@lin projem.init]$ ll
total 32
drwxr-x--- 2 oacar users 4096 Apr  7 13:36 branches
-rw-r----- 1 oacar users   66 Apr  7 13:36 config
-rw-r----- 1 oacar users   73 Apr  7 13:36 description
-rw-r----- 1 oacar users   23 Apr  7 13:36 HEAD
drwxr-x--- 2 oacar users 4096 Apr  7 13:36 hooks
drwxr-x--- 2 oacar users 4096 Apr  7 13:36 info
drwxr-x--- 4 oacar users 4096 Apr  7 13:36 objects
drwxr-x--- 4 oacar users 4096 Apr  7 13:36 refs
[oacar@lin projem.init]$
```

Yalın bir depo oluşturmak için --bare parameresini kullanmamız gerekmektedir. Yalın depoların isimleri .git ekini taşımaktadırlar. Şema 19 da yer alan örnekte projem.git ismini taşıyan yalın bir depo oluşturduk. Bu depoyu kullanarak, yeni bir çalışma dizini oluşturmak için depoyu klonlamamız gerekiyor. Yalın depolara doğrudan dosya eklemek mümkün değildir. Bu yüzden yeni bir klon oluşturmamız gerekmektedir.

Git Clone

Clone komutu ile mevcut bir Git deposunun kopyasını şu şekilde oluşturabiliriz:

```
// Şema 20

[oacar@lin repo]$ git clone projem.git/ projem-clone
Cloning into 'projem-clone'...
warning: You appear to have cloned an empty repository.
done.
[oacar@lin repo]$ ls -l
total 8
drwxr-x--- 3 oacar users 4096 Apr  7 13:42 projem-clone
drwxr-x--- 7 oacar users 4096 Apr  7 13:41 projem.git
```

Boş ve yalın bir deponun (projem.git) projem-clone isminde bir kopyasını oluşturduk. Şimdi oluşturduğumuz bu yeni klon depoya yeni bir dosya ekleyelim. Şema 21 de bunun nasıl yapıldığı yer almaktadır.

```
// Şema 21

[oacar@lin repo]$ cd projem-clone/

[oacar@lin projem-clone]$ touch test.txt

[oacar@lin projem-clone]$ echo "test" > test.txt

[oacar@lin projem-clone]$ git status
On branch master

Initial commit

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

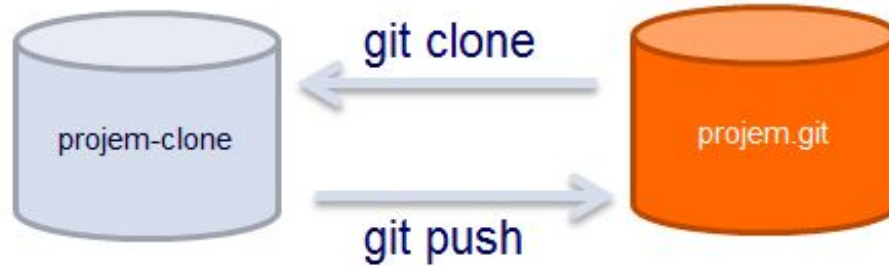
    test.txt

nothing added to commit but untracked files present (use "git add" to track)

[oacar@lin projem-clone]$ git add --all

[oacar@lin projem-clone]$ git commit -m "initial commit"
[master (root-commit) 2f5e008] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
[oacar@lin projem-clone]$
```

Klon depo (projem-clone) üzerinde yaptığımız değişiklikleri commit ile yeni depoya eklemiş olduk. Klon depo ile orijinal depo arasındaki ilişkiyi resim 3 de görmekteyiz.



Resim 3

Yaptığımız bu değişiklikler commit komutu ile klon depoya eklenmiş olmasına rağmen, bu değişikliklerin push komutu ile istenildiği taktirde orijinal depoya (projem.git) aktarılması mümkündür. Eğer birden fazla kullanıcı projem.git deposu ile çalışıyorsa, yani projem.git merkezi bir Git deposuysa, o taktirde klon depolarda yapılan değişikliklerin push komutu ile orijinal depoya aktarılmasında fayda vardır.

Push komutunu şu şekilde kullanabiliriz:

```
// Şema 22

[oacar@lin projem-clone]$ git push

No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
fatal: The remote end hung up unexpectedly
error: failed to push some refs to '/export/home/oacar/repo/projem.git/'
[oacar@lin projem-clone]$
```

Orijinal depo bünyesinde master ismini taşıyan bir dal (branch) olmadığı hatasını aldık. Dal konseptini daha sonra yakından inceleyeceğiz. İçi boş olan yalın depolarda master isminde bir yapı olmadığından bu hatayı aldık. Push komutunu aşağıdaki şekilde değiştirerek, iki depo arasındaki senkronizasyon işlemini tekrar edebiliriz.

```
// Şema 23

[oacar@lin projem-clone]$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 219 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
To /export/home/oacar/repo/projem.git/
 * [new branch]      master -> master
[oacar@lin projem-clone]$
```

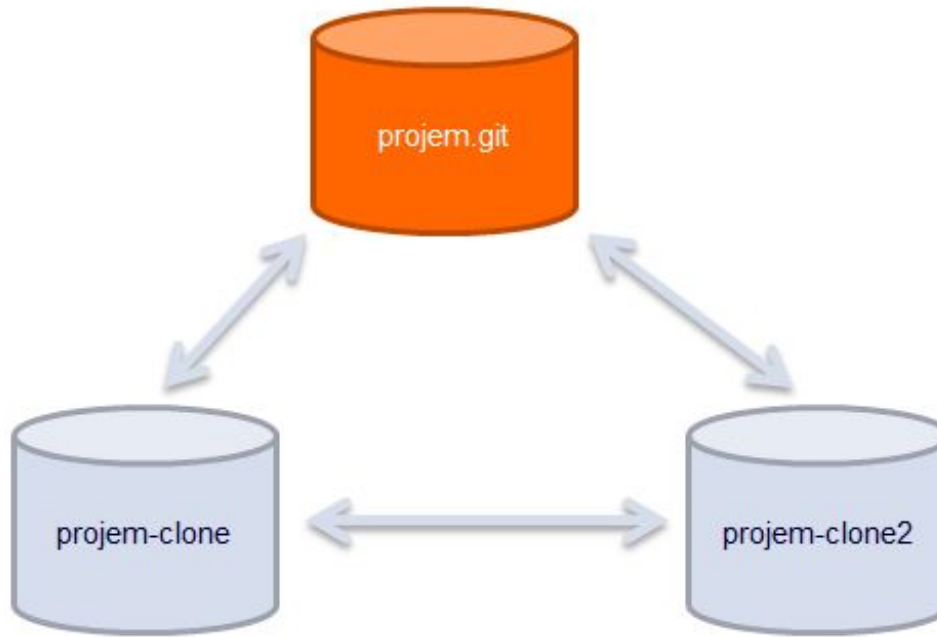
Push komutunda kullandığımız origin merkezi depoya işaret eden bir isim, master da bu depodaki master dalıdır. Kitabın dördüncü bölümünde origin konseptini daha yakından inceleyeceğiz. Klon depo üzerinde yaptığımız değişikliklerin orijinal depoya aktarıldığından emin olmak için orijinal depoyu başka bir isim altında tekrar klonluyoruz.

```
// Şema 24

[oacar@lin repo]$ git clone projem.git/ projem-clone2
Cloning into 'projem-clone2'...
done.
[oacar@lin repo]$ cd projem-clone2/
[oacar@lin projem-clone2]$ ll
total 4
-rw-r----- 1 oacar users 5 Apr  7 15:07 test.txt
[oacar@lin projem-clone2]$
```

Şema 24 de görüldüğü gibi klon depo bünyesinde oluşturduğumuz test.txt dosyası push komutu ile orijinal depoya aktarılmıştır. Eğer push işlemi gerçekleşmemiş olsaydı, şema 24 de yeni oluşturduğumuz ikinci klon depo bünyesinde test.txt dosyasını göremezdik.

Kendisi klon olan bir depodan yeni bir klon oluşturarak, bu iki depo arasında klon-origin ilişkisi tanımlanabilmektedir. Git dünyasında aslında merkezi bir deponun gerekli olmadığını, depolar arasında veri alışverişinin push ve pull komutlarıyla yapılacağını resim 4 de görmekteyiz. Kitabın üçüncü ve dördüncü bölümlerinde Git'in bu dağıtık (distributed) ve merkezi olmayan (decentral) yapısını yakından inceleyeceğiz.



Resim 4

Git Pull

Git clone komutu ile bir kopyasını oluşturduğumuz bir depoya, klon üzerinde yaptığımız değişiklikleri eklemek için git push komutu kullanmamız gerektiğini görmüştük. Aynı şekilde üst depoda oluşan değişiklikleri kendi klon depomuza alabilmek için git pull komutunu kullanabiliriz. Bunun nasıl yapıldığını ve ne zaman gerekli olduğunu şimdi bir örnek üzerinde inceleyelim:

```
[oacar@lin repo]$ git clone projem.git/ klon1
Cloning into 'klon1'...
done.
[oacar@lin repo]$ git clone projem.git/ klon2
Cloning into 'klon2'...
done.
```

Üst depomuz projem.git ismini taşıyor ve git clone ile klon1 ve klon2 isminde iki klon depo oluşturuyoruz. Şimdi klon1 ve klon2 içinde neler olduğuna bir göz atalım:

```
[oacar@lin repo]$ cd klon2/

[oacar@lin klon2]$ ls -l
total 8
-rw-r----- 1 oacar users 5 Apr 20 13:56 test.txt
-rw-r----- 1 oacar users 4 Apr 20 13:56 xxx
```



```
[oacar@lin klon2]$ cd ..

[oacar@lin repo]$ cd klon1/

[oacar@lin klon1]$ ls -l
total 8
-rw-r----- 1 oacar users 5 Apr 20 13:56 test.txt
-rw-r----- 1 oacar users 4 Apr 20 13:56 xxx
```

İçerik olarak iki depo birbirinin kopyası, çünkü projem.git isimli depodan oluşturuldular. Şimdi klon2 içinde değişiklik yapıp, bu değişiklikleri üst depo olan projem.git e gönderelim:

```
[oacar@lin repo]$ cd klon2/

[oacar@lin klon2]$ touch abc

[oacar@lin klon2]$ git add abc

[oacar@lin klon2]$ git commit -m "abc eklendi"
[master 83ce168] abc eklendi
1 file changed, 1 insertion(+)
create mode 100644 abc

[oacar@lin klon2]$ git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 307 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /export/home/oacar/repo/projem.git/
4431e58..83ce168 master -> master
```

Şimdi klon1 e göz atalım:

```
[oacar@lin klon2]$ cd ..

[oacar@lin repo]$ cd klon1/

[oacar@lin klon1]$ ls -l
total 8
-rw-r----- 1 oacar users 5 Apr 20 13:56 test.txt
-rw-r----- 1 oacar users 4 Apr 20 13:56 xxx
```

İçerikte bir değişiklik yok, yalnız üst depo olan projem.git de bir değişiklik oldu. Bu değişikliği

almak için git pull komutunu şu şekilde kullanıyoruz:

```
[oacar@lin klon1]$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /export/home/oacar/repo/projem
 4431e58..83ce168  master    -> origin/master
Updating 4431e58..83ce168
Fast-forward
 abc | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 abc
```

Pull işleminden sonra tekrar klon1 in içeriğine bakıyoruz:

```
[oacar@lin klon1]$ ls -l
total 12
-rw-r----- 1 oacar users 9 Apr 20 14:00 abc
-rw-r----- 1 oacar users 5 Apr 20 13:56 test.txt
-rw-r----- 1 oacar users 4 Apr 20 13:56 xxx
[oacar@lin klon1]$
```

Görüldüğü gibi klon2 aracılığı ile yaptığımız değişikliği push komutu ile projem.git e eklendikten sonra, bu değişikliği klon1 bünyesinde pull komutu ile edindik. Bu şekilde depolar arasında her iki yönde yapılan değişiklikleri push ve pull komutları ile edinebileceğimizi görmüş olduk.

Git Log

Her commit işlemi ile Git deposu üzerinde değişiklikler yapılır. Bu değişiklikleri takip etmek için git log komutunu şu şekilde kullanabiliriz:

```
// Şema 25

[oacar@lin repol]$ git log
commit 9bde873b2ff837cf2bc6795b3f5870a88ef5abe4
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:19:34

    dördüncü commit

commit 9678fa24ee3268d996dd0cdde79e1076c1089d9f
```

```

Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:26

    üçüncü commit

commit f09bc20f86e5ef17bae09e3f8ec40c01dce83354
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:14

    ikinci commit

commit e2b58594b252785273866fd4e82e909a4e60bb26
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:15:01

    ilk commit
[oacar@lin repol]$

```

Şema 25 de yer alan listeyi aşağıdan yukarıya doğru okuduğumuzda, depo üzerinde yapılan işlemlerin tarihçesini (history) görebiliriz. Buna göre ilk commit e2b58594b252785273866fd4e82e909a4e60bb26, son commit 9bde873b2ff837cf2bc6795b3f5870a88ef5abe4 değerini taşımaktadır. Bu harf ve rakamdan oluşan yapıya commit hash ismi verilmektedir. Git hash değerini commit esnasında yapılan değişiklikleri göz önünde bulundurarak hesaplamaktadır. Commit hash değeri değişikliğe uğrayan tüm dosyalar üzerinde SHA-1 algoritması ile yapılan hesaplama işleminin (checksum) sonucudur. Bu hash değeri ile hem commit ile değişikliğe uğrayan dosyaların başkaları tarafından sonradan degistirilmemeleri için önlem alınmaktadır hem de commit için tekil bir anahtar oluşturulmaktadır.

Commit hash değerini kullanarak, commit bünyesinde yer alan değişikliklere ulaşabiliriz. Bunun için tüm hash değerini kullanmamız gerekmiyor. Bunun yerine ilk yedi hane yeterli olmaktadır. Örneğin ilk commitin içeriğini şu şekilde edinebiliriz:

```

// Şema 26

[oacar@lin repol]$ git show e2b58594
commit e2b58594b252785273866fd4e82e909a4e60bb26
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:15:01

    ilk commit

diff --git a/test1 b/test1
new file mode 100644
index 0000000..e69de29

```

```
[oacar@lin repol]$
```

İlk commit hakkında aldığımız bilgede (şema 26) commit mesajını (ilk commit) ve hangi dosyanın (test1) değişikliğe uğradığınızı görmekteyiz. Son commite göz atalım:

```
// Şema 27

[oacar@lin repol]$ git show 9bde873b2
commit 9bde873b2ff837cf2bc6795b3f5870a88ef5abe4
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:19:34

    dördüncü commit

diff --git a/test3 b/test3
index e69de29..9daefb 100644
--- a/test3
+++ b/test3
@@ -0,0 +1 @@
+test
[oacar@lin repol]$
```

Son commit bünyesinde test3 isimli dosyanın değişikliğe uğradığını görmekteyiz. Burada +1 test3 dosyasına yeni bir satırın eklendiğini, +test ise eklenen satırı göstermektedir. Artı işareti dosyalara eklenen, eksi işareti dosyalardan silinen satırları göstermektedir.

Git log komutu ile ekran çıktısını değişik türde şekillendirmek mümkündür. Örneğin tüm commitleri liste olarak almak istersek:

```
[oacar@lin repol]$ git log --oneline
9bde873 dördüncü commit
9678fa2 üçüncü commit
f09bc20 ikinci commit
e2b5859 ilk commit
[oacar@lin repol]$
```

Son üç commiti almak istersek:

```
[oacar@lin repol]$ git log -n 3 --oneline
35f95ab altıncı commit
80b8fc4 beşinci commit
9bde873 dördüncü commit
[oacar@lin repol]$
```

Hangi dosyaların ne oranda değişikliğe uğradıklarını görmek istersek:

```
[oacar@lin repol]$ git log --stat
commit 9bde873b2ff837cf2bc6795b3f5870a88ef5abe4
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:19:34

    dördüncü commit

test3 | 1 +
1 file changed, 1 insertion(+)

commit 9678fa24ee3268d996dd0cdde79e1076c1089d9f
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:26

    üçüncü commit

test3 | 0
1 file changed, 0 insertions(+), 0 deletions(-)

commit f09bc20f86e5ef17bae09e3f8ec40c01dce83354
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:14

    ikinci commit

test2 | 0
1 file changed, 0 insertions(+), 0 deletions(-)

commit e2b58594b252785273866fd4e82e909a4e60bb26
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:15:01

    ilk commit

test1 | 0
1 file changed, 0 insertions(+), 0 deletions(-)
[oacar@lin repol]$
```

Dosyaların içeriklerinin nasıl değiştiğini görmek istersek:

```
[oacar@lin repol]$ git log -p
commit 9bde873b2ff837cf2bc6795b3f5870a88ef5abe4
Author: Özcan <acar@agilementor.com>
```

```
Date:   Mon Apr 20 15:19:34
```

```
    dördüncü commit
```

```
diff --git a/test3 b/test3
index e69de29..9daeafb 100644
--- a/test3
+++ b/test3
@@ -0,0 +1 @@
+test
```

```
commit 9678fa24ee3268d996dd0cdde79e1076c1089d9f
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:26
```

```
    üçüncü commit
```

```
diff --git a/test3 b/test3
new file mode 100644
index 0000000..e69de29
```

```
commit f09bc20f86e5ef17bae09e3f8ec40c01dce83354
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:14
```

```
    ikinci commit
```

```
diff --git a/test2 b/test2
new file mode 100644
index 0000000..e69de29
```

```
commit e2b58594b252785273866fd4e82e909a4e60bb26
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:15:01
```

```
    ilk commit
```

```
diff --git a/test1 b/test1
new file mode 100644
index 0000000..e69de29
[ocar@lin repol]$
```

Belli bir kullanıcı tarafından yapılan commitleri görmek istersek:

```
[ocar@lin repol]$ git log --author="acar" --oneline
35f95ab altıncı commit
```

```
80b8fc4 beşinci commit
9bde873 dördüncü commit
9678fa2 üçüncü commit
f09bc20 ikinci commit
e2b5859 ilk commit
[oacar@lin repol]$
```

Belli bir isme sahip dosyanın dahil olduğu commitleri görmek istersek:

```
[oacar@lin repol]$ git log test2
commit f09bc20f86e5ef17bae09e3f8ec40c01dce83354
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 20 15:16:14

    ikinci commit
[oacar@lin repol]$
```

Daha kısa ekran çıktısı almak için --oneline parametresini kullanabiliriz. Aşağıda test3 isimli dosyanın yer aldığı commit listesini görmekteyiz.

```
[oacar@lin repol]$ git log --oneline test3
35f95ab altıncı commit
80b8fc4 beşinci commit
9bde873 dördüncü commit
9678fa2 üçüncü commit
[oacar@lin repol]$
```

Son Commitin Değiştirilmesi

Oluşturduğumuz bir commite bir dosyayı eklemeyi unuttuğumuzu düşünelim. Bu dosyayı en son commite ekleyebiliriz yani oluşturduğumuz en son commitin içeriğini değiştirebiliriz. Bunun için --amend parametresini aşağıdaki şekilde kullanabiliriz.

Önce yeni bir commit oluşturuyoruz:

```
[oacar@lin projem2]$ touch dosya1

[oacar@lin projem2]$ git add dosya1

[oacar@lin projem2]$ git commit -m "ilk commit"
[master (root-commit) 367bca2] ilk commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya1
```

Commit listemiz şu şekilde oldu:

```
[oacar@lin projem2]$ git log --oneline
367bca2 ilk commit
```

Git status ile son durumu alıyoruz:

```
[oacar@lin projem2]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

Şimdi yeni bir dosya oluşturalım:

```
[oacar@lin projem2]$ touch dosya2

[oacar@lin projem2]$ git add dosya2

[oacar@lin projem2]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 24 14:43 dosya1
-rw-r----- 1 oacar users 0 Apr 24 14:46 dosya2
```

Bu dosyanın aslında ilk commit içinde olması gerektiğini varsayalım. Bu durumda dosya2 isimli dosyayı şu şekilde 367bca2 hash değerindeki commite dahil edebiliriz:

```
[oacar@lin projem2]$ git commit --amend
```

Git commit mesajını değiştirmek için 367bca2 hash değerindeki ilk commitin içeriğini bir editörde açacaktır ve içerik şu şekilde olacaktır:

```
ilk commit

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   dosya1
```



```
#      new file:   dosya2
#
```

İstersek commit mesajını değıtirebiliriz. Bunun yanı sıra commit bünyesinde yer alacak değışiklikleri görmekteyiz. Editörü kapattıktan sonra ekran çıktısı şu şekilde olacaktır:

```
[oacar@lin projem2]$ git commit --amend
[master 919cdd0] ilk commit
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya1
create mode 100644 dosya2
```

Şimdi tekrar commit listesine bakalım:

```
[oacar@lin projem2]$ git log --oneline
919cdd0 ilk commit
```

Görüldüğü gibi listede tek bir commit var. 367bca2 hash değıerindeki commiti göremiyoruz. Bunun yerine 919cdd0 hash değıerinde yeni bir commit geldi. Bu aslında eski 367bca2 hash değıerindeki commit, lakin içeriğı değıştiğı için hash değıeri de değışti. Commitin içeriğıne şu şekilde göz atabiliriz:

```
[oacar@lin projem2]$ git show 919cdd0
commit 919cdd05bae8a5b4215f02733a92b58431a8b892
Author: Özcan <acar@agilementor.com>
Date:   Fri Apr 24 14:45:19

    ilk commit

diff --git a/dosya1 b/dosya1
new file mode 100644
index 0000000..e69de29
diff --git a/dosya2 b/dosya2
new file mode 100644
index 0000000..e69de29
[oacar@lin projem2]$
```

Görüldüğü gibi yeni commit her iki dosyayı da ihtiva etmektedir.

Git Revert ile Değışikliklerin Geri Alınması

Amend parametresi ile mevcut bir commit üzerinde değışiklik yapabilirken, git revert komutu ile

mevcut bir commit bünyesindeki değişiklikleri geriye çeviren yeni bir commit oluşturulmaktadır. Bunun nasıl yapıldığını bir örnek üzerinde inceleyelim:

Önce deponun commit listesine bir göz atalım:

```
[oacar@lin projem]$ git log --oneline
417cled ikinci commit
2d8bf77 ilk commit
```

Şimdi yeni bir dosya oluşturup, commitleyelim:

```
[oacar@lin projem]$ touch dosya5

[oacar@lin projem]$ git add dosya5

[oacar@lin projem]$ git commit -m "üçüncü commit"
[master 26916ee] üçüncü commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya5
```

Yeni commit listesi şu şekilde olacaktır:

```
[oacar@lin projem]$ git log --oneline
26916ee üçüncü commit
417cled ikinci commit
2d8bf77 ilk commit
[oacar@lin projem]$
```

En son commit 26916ee hash değerini taşıyor. Bu commitin içeriğine su şekilde bakabiliriz:

```
[oacar@lin projem]$ git log --oneline
26916ee üçüncü commit
417cled ikinci commit
2d8bf77 ilk commit

[oacar@lin projem]$ git show 26916ee
commit 26916ee0114470c7334796e14ca2e886319c9faa
Author: Özcan <acar@agilementor.com>
Date:   Fri Apr 24 15:09:20

    üçüncü commit

diff --git a/dosya5 b/dosya5
new file mode 100644
```

```
index 0000000..e69de29
```

26916ee hash değerindeki commit ile dosya5 isimli dosyanın oluşturulduğunu görmekteyiz. Bu değişikliği geri almak istediğimizi düşünelim yani dosya5 dosyasının ve yaptığımız diğer tüm değişikliklerin depodan silinmesini istiyoruz. Bu durumda revert komutu şu şekilde kullanabiliriz:

```
[oacar@lin projem]$ git revert 26916ee
[master cefba26] Revert "üçüncü commit"
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 dosya5
```

Revert komutu ile dosya5 dosyası hem depodan hem de içinde çalıştığımız dizinden (working copy) silindi. Tekrar commit listesine bir göz atalım:

```
[oacar@lin projem]$ git log --oneline
cefba26 Revert "üçüncü commit"
26916ee üçüncü commit
417cled ikinci commit
2d8bf77 ilk commit
```

Görüldüğü gibi cefba26 hash değerinde ve yapılan değişiklikleri ihtiva eden yeni bir commit oluştu. Bu commit aslında 26916ee öncesindeki duruma geri dönüşü sembolize etmektedir. İçeriğine baktığımızda şunları görmekteyiz:

```
[oacar@lin projem]$ git show cefba26
commit cefba26b76977df189cd78c5f2806d7f70b936d3
Author: Özcan <acar@agilementor.com>
Date:   Fri Apr 24 15:12:30

    Revert "üçüncü commit"

    This reverts commit 26916ee0114470c7334796e14ca2e886319c9faa.

diff --git a/dosya5 b/dosya5
deleted file mode 100644
index e69de29..0000000
```

Sondan üçüncü satırda dosya5 isimli dosyanın silindiğini görmekteyiz. Eğer aynı zamanda diğer dosyalar üzerinde değişiklik yapmış olsaydık, revert ile bu değişiklikler de geri alınırdı, yani commit öncesindeki haline geri getirilirlerdi.

Şimdi içinde çalıştığımız dizinin yapısı şöyle:

```
[oacar@lin projem]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 24 15:08 dosya1
-rw-r----- 1 oacar users 0 Apr 24 15:08 dosya2
-rw-r----- 1 oacar users 0 Apr 24 15:08 dosya3
-rw-r----- 1 oacar users 0 Apr 24 15:08 dosya4
```

Görüldüğü gibi dosya5 hem dizinden hem de depodan silindi. Revert komutunun tek dejavantajı, yapılan değişikliklerin geri alınmasını sağlamakla birlikte, bu değişikliklerin izlerini commit listesinde bırakmasıdır. Git revert bunun yanı sıra geri alınan değişiklikleri göstermek amacıyla yeni bir revert commit oluşturmaktadır.

Etiket Kullanımı

Yazılımda gelinecek belli aşamaları işaretlemek için etiketler kullanılır. Etiketleri yazılım esnasında oluşturulan kilometre taşları olarak düşünebiliriz. Etiket isimleri aracılığı ile geçmişte sağlanan proje aşamalarına geri dönmek mümkündür.

Etiketler çoğunlukla yeni sürümler (release) oluşturulurken, sürümün kapsadığı kodu işaretlemek için kullanılır. Bu bölümde etiketlerin nasıl oluşturulduğunu ve kullanıldığını örnekler üzerinde inceleyeceğiz.

Bir depo bünyesinde yer alan etiketleri şu şekilde elde edebiliriz:

```
[oacar@lin projem]$ git tag
v1.0.0
v1.1.0
v2.0.0
```

Yeni bir etiketi şu şekilde oluşturabiliriz:

```
[oacar@lin projem]$ git tag -a v3.0.0 -m "3.0.0 version created"
[oacar@lin projem]$ git tag
v1.0.0
v1.1.0
v2.0.0
v3.0.0
```

Kullandığımız `-m` parametresi ile etikete açıklayıcı bir metin ekleyebiliriz. Bu parametreyi kullanmadığımız takdirde, git bir editör açarak, böyle bir metni girmemizi talep edecektir.

Show komutunu kullanarak, etiketin hangi commiti baz aldığını öğrenebiliriz:

```
[oacar@lin projem]$ git tag -a v3.0.0 -m "3.0.0 version created"
[oacar@lin projem]$ git show v3.0.0
tag v3.0.0
Tagger: Özcan Acar <oezcanacar@xxx>
Date:   Tue Aug 11 14:38:08 2015 +0200

3.0.0 version created

commit b17908598c36f52eeb11f66cf334c0831f736ee5
Author: Özcan Acar <oezcanacar@xxx>
Date:   Tue Aug 11 14:31:29 2015 +0200

    yeni bir commit.

diff --git a/x b/x
new file mode 100644
index 0000000..e69de29
```

v3.0.0 ismini taşıyan etiket b179085 hash değerine sahip commiti baz almıştır. Git bu şekilde oluşturulan etiketi güncel commit ile ilişkilendirmektedir. Commit hash değeri uzun bir rakam ve harf kombinasyonu iken, etiket isimlerini daha akılda kalıcı şekilde seçerek, projenin belli aşamalarını işaretleyebiliriz.

Git'in sunduğu ikinci bir alternatif ile basit etiket olarak bilinen etiketler oluşturmak mümkündür. Basit etiketler oluştururken sadece etiket ismini tanımlamak yeterlidir. Git güncel commit hash değerini kullanarak, bu etiketi oluşturur. Bunun bir örneğini aşağıda görmekteyiz.

```
[oacar@lin projem]$ git tag v4.0.0

[oacar@lin projem]$ git tag
v1.0.0
v1.1.0
v2.0.0
v3.0.0
v4.0.0

[oacar@lin projem]$ git show v4.0.0
commit b17908598c36f52eeb11f66cf334c0831f736ee5
Author: Özcan Acar <oezcanacar@xxx>
Date:   Tue Aug 11 14:31:29 2015 +0200

    yeni bir commit.
```

```
diff --git a/x b/x
new file mode 100644
index 0000000..e69de29
```

Git etiketi oluştururken mevcut commiti baz olarak alıyor demiştik. Bir etiket oluşturmak için kullanabileceğimiz tek yöntem bu değil. Geçmişte oluşturduğumuz commitleri kullanarak da etiket oluşturabiliriz. Bunun nasıl yapıldığını şimdi bir örnek üzerinde inceleyelim.

Öncelikle commit listesini ediniyoruz:

```
[oacar@lin projem]$ git log --pretty=oneline
b17908598c36f52eeb11f66cf334c0831f736ee5 test2.txt eklendi
55962fccf408cbe26a9b29e2724b7e267f305fa9 test1.txt eklendi
3220c66113b53eb2f387f7c2b50f2842564829fa ilk commit
```

Örneğin "test1.txt eklendi" commitinden sonra v1.2.0 isimli etiketi oluşturmayı unuttuğumuzu düşünelim. 55962fc hash değerine sahip commiti baz alan etiketi şu şekilde oluşturabiliriz:

```
[oacar@lin projem]$ git tag -a v1.2.0 -m 'version 1.2.0' 55962fc
```

```
[oacar@lin projem]$ git tag
v1.0.0
v1.1.0
v1.2.0
v2.0.0
v3.0.0
v4.0.0
```

```
[oacar@lin projem]$ git show v1.2.0
tag v1.2.0
Tagger: Özcan Acar <oezcanacar@xxx>
Date:   Tue Aug 11 15:04:25 2015 +0200
```

```
version 1.2.0
```

```
commit 55962fccf408cbe26a9b29e2724b7e267f305fa9
Author: Özcan Acar <oezcanacar@xxx>
Date:   Tue Aug 11 13:25:39 2015 +0200
```

```
test1 eklendi
```

```
diff --git a/test1 b/test1
new file mode 100644
index 0000000..e69de29
```

Daha öncede belirttiğim gibi etiketler yerel depoda oluşturulmaktadır. Uzakta (remote) bulunan bir depoya bir ekiletleri su şekilde aktarabiliriz:

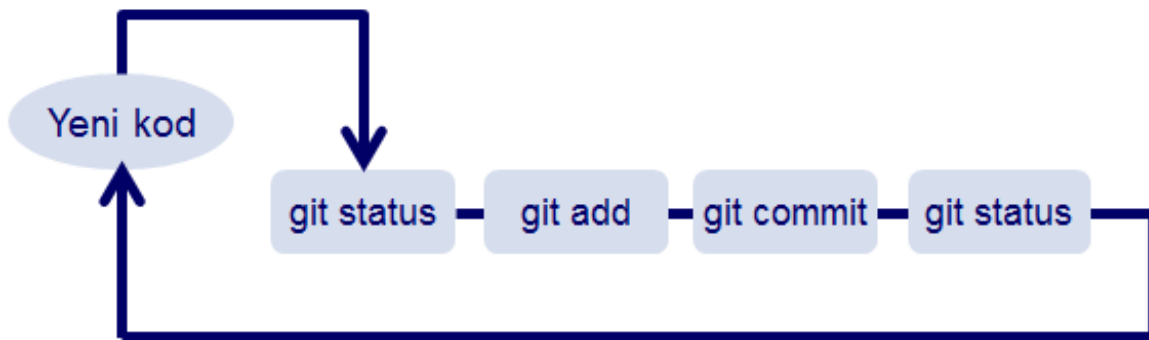
```
[oacar@lin projem]$ git push origin --tags
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 682 bytes | 0 bytes/s, done.
Total 7 (delta 3), reused 0 (delta 0)
To https://github.com/ozcanacar/pratik-git.git
* [new tag]          show -> show
* [new tag]          v1.0.0 -> v1.0.0
* [new tag]          v1.1.0 -> v1.1.0
* [new tag]          v1.2.0 -> v1.2.0
* [new tag]          v2.0.0 -> v2.0.0
* [new tag]          v3.0.0 -> v3.0.0
* [new tag]          v4.0.0 -> v4.0.0
```

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Yeni bir Git deposu oluşturmak için `git init` komutu kullanılır.
- Depoya yeni dosya eklemek için kullanılan komut `git add` komutudur.
- `Git rm` komutu ile depodaki mevcut bir dosya silinebilir.
- Yalın bir Git deposu oluşturmak için `git init` komutuna `--bare` parametresi eklenir.
- `Git commit` komutu ile dosyalar üzerinde yapılan değişiklikler depoya aktarılır.
- Mevcut bir depoyu klonlamak için `git clone` komutu kullanılır.
- Depolar arası senkronizasyon için `git push/pull` komutları kullanılır. Push alt depodaki değişiklikleri üst depoya aktarırken, pull üst depo değişikliklerini alt depoya çekmektedir.
- Depoya eklenmemesini istemediğimiz dosyalar için `.gitignore` dosyası gereklidir. Bu dosyada depoya eklenmemesi gereken dosya isimleri yer alır.
- `Git config` komutu ile Git kurulumu ve depolar için gerekli genel ve kullanıcı bazlı ayarları yapmak mümkündür.
- `Git log` komutu ile commitler bünyesinde yapılan değişiklikleri edinebiliriz.
- `Git commit --amend` ile son commit nesnesi için girilen mesajı değiştirebiliriz.
- `Git revert` komutu ile bir commit bünyesinde değişikliğe uğrayan dosyaların eski hallerine alınması mümkündür.
- `Git tag` komutu ile mevcut etiketleri edinebilir ve yeni etiketler oluşturabiliriz.

Tipik bir Git iş akışı resim 5 de görülmektedir. Git kullanıcıları günlük işlerinde bu döngü içinde gerekli Git işlemlerini gerçekleştirirler.



Resim 5

3. Bölüm

Git Dal Yönetimi

Branch Konsepti

Branch (dal ya da kol) konseptini kullanarak, birbirinden bağımsız uygulama özelliklerini paralel olarak implemente edebiliriz. Bunun nasıl yapıldığını bir örnek üzerinde inceleyelim.

Üzerinde çalıştığımız projenin yapısı aşağıdaki şekildedir:

```
// Şema 1

[oacar@lin projem-klon]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 21 13:12 dosya1
-rw-r----- 1 oacar users 0 Apr 21 13:13 dosya2
-rw-r----- 1 oacar users 0 Apr 21 13:13 dosya3
[oacar@lin projem-klon]$
```

Kullandığımız Git deposunun commit listesi şu şekildedir:

```
// Şema 2

[oacar@lin projem-klon]$ git log --oneline
43750d9 üçüncü commit
a000068 ikinci commit
9c59180 ilk commit
```

Depo üzerinde yapılan işlemleri şu şekilde görselleştirebiliriz:



Resim 1

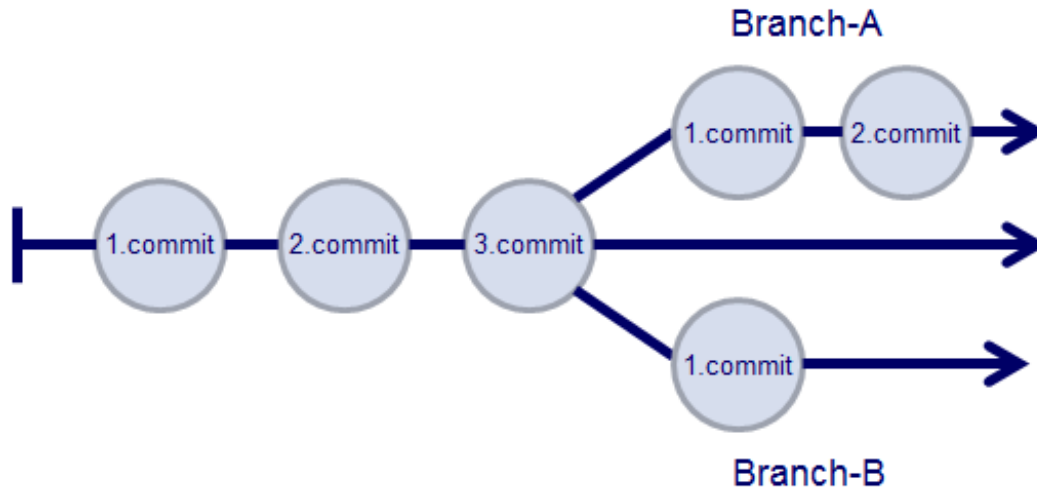
Resim 1 de görüldüğü gibi arka arkaya üç adet commit yapılmıştır. Her commit kendinden önceki commiti başı (parent) olarak tanımaktadır. Buradaki tek istisnayı birinci commit teşkil etmektedir. Birinci commitin başı bulunmamaktadır, yani null değerindedir.

Bu yapıda henüz dal konseptiyle tanışmış olmasak bile Git'in sunduğu ana dalı görmekteyiz. Resim 1 de yer alan commitlerin hepsi master ismini taşıyan dal üzerinde yer almaktadır. Her commiti bir kilometre taşı olarak düşünebiliriz. Herhangi iki kilometre taşı arasındaki farklılığı deponun geçmişine bakarak, tespit edebiliriz. Her commit ile projeye yeni bir kilometre taşı atamış

olmaktayız. Her commit bazı dosyalar üzerinde modifikasyon yaptığımız anlamına gelmektedir. Bu projenin ilerlemesi açısından gerekli bir durum olmakla birlikte, değişik uygulama özelliklerini paralel implemente eden ekiplerde sorun olabilmektedir. Bunun nedenini bir örnek ile üzerinde açıklamak istiyorum.

Proje ve depo yapımız şema 1 deki şekilde olsun. Bizden ekip olarak bir hafta sonra A ve B isimlerini taşıyan iki yeni uygulama özelliğini ihtiva eden yeni bir sürüm oluşturmamız istendi. Yani bir hafta içinde bu iki özelliği implemente etmemiz gerekiyor. Bir haftalık zaman diliminde sadece A ismini taşıyan özelliği implemente ettiğimizi düşünelim. B özelliği ne yazık ki yarım kaldı. Bu zaman zarfında depo üzerinde hem A için hem de B için birçok commit yapıldı. Bir hafta sonra master dalında yer alan kodu kapsayan yeni bir sürüm (örneğin bir Java web uygulamasında bu bir .war dosyasıdır) oluşturduğumuzda, sürüm paketinin içinde tamamladığımız A ama aynı zamanda tamamlamadığımız B değişiklikleri de yer alacaktır. Bu normal şartlarda istenmeyen bir durumdur. Tamamlanmayan özelliklerin sürüm içinde yer alması, uygulamanın davranış biçimini olumsuz etkileyebilir. Bu yüzden uygulamanın değişik parçalarını birbirlerinden bağımsız ve paralel olarak implemente etmek için başka bir çözüm bulmamız gerekiyor. Bu çözüm her yeni uygulama özelliği için bir branch yani dal oluşturmaktır.

Şimdi bize verilen görevi dal konseptini uygulayacak şekilde tekrar gözden geçirelim. Bir hafta içinde A ve B ismini taşıyan iki yeni özellik implemente etmek istiyoruz. A örneğin kullanıcının isim ve şifresi ile sisteme girişi, B ise profil bilgilerinin gösterilme işlemi olsun. Birbirinden bağımsız bu iki uygulama özelliğini birbirlerine paralel, yani birbirlerini etkilemeyecek şekilde implemente etmek için Branch-A ve Branch-B isimlerini taşıyan iki yeni dal oluşturmuyoruz. Master haricinde oluşturduğumuz her yeni dalın mevcut bir commiti baz alması gerekmektedir. Seçtiğimiz commitden yola çıkarak yeni bir dal oluşturduğumuzda, ana dal olan master ile yollarımızı ayırmış oluyoruz. Oluşturduğumuz yeni dal üzerinde çalışmaya başladığımız andan itibaren master üzerinde hiçbir değişiklik yapılmayacak, tüm değişiklikler üzerinde çalıştığımız dal üzerinde gerçekleşecektir.



Resim 2

Resim 2 de görüldüğü gibi A ve B birbirlerinden bağımsız ve master dalı etkilemeyecek şekilde geliştirilebilir hale gelmişlerdir. Branch-A ve Branch-B master üzerindeki üçüncü commiti baz alarak, yollarını masterdan ayırmışlardır. Branch-A dalında A için iki commit, Branch-B dalında B için bir commit görülmektedir. Eğer A ve B için dal konseptini uygulamamış olsaydık, A ve B için yapılan commitler ana dal olan master üzerinde yer alacaklardı ve sürümün bir parçası haline geleceklerdi. Ama implemente etmek istediğimiz özellikleri dal kullanımı yardımı ile izole ederek, sürümün içeriğini de kontrol edilebilir hale getirmiş olduk. Şimdi dal konseptinin Git bünyesinde nasıl uygulandığını yakından inceleyelim.

Kullandığımız git deposundaki dal listesini şu şekilde edinebiliriz:

```
[oacar@lin projem-klon]$ git branch
* master
[oacar@lin projem-klon]$
```

Görüldüğü gibi depomuzda sadece ana dal olan master dalı bulunmaktadır. Yeni bir dalı şu şekilde oluşturabiliriz:

```
// Yeni dalı oluşturur
[oacar@lin projem-klon]$ git branch branch-a

// Dal listesini verir
[oacar@lin projem-klon]$ git branch
  branch-a
* master
[oacar@lin projem-klon]$
```

Branch-a ismini taşıyan yeni dal oluşturduk. Bu dalı şu şekilde silebiliriz:

```
// Mevcut dal listesi
[oacar@lin projem-klon]$ git branch
  branch-a
* master
[oacar@lin projem-klon]$

// Branch-a dalı siliniyor
[oacar@lin projem-klon]$ git branch -d branch-a
Deleted branch branch-a (was 43750d9).

// Yeni dal listesi
[oacar@lin projem-klon]$ git branch
* master
[oacar@lin projem-klon]$
```

Oluşturduğumuz yeni dal üzerinde çalışabilmek için checkout komutu ile bu dala geçmemiş gerekiyor. Bunu şu şekilde yapabiliriz:

```
// Branch-a dalına geçiş
[oacar@lin projem-klon]$ git checkout branch-a
Switched to branch 'branch-a'

// Dal listesi
[oacar@lin projem-klon]$ git branch
* branch-a
  master
[oacar@lin projem-klon]$
```

Görüldüğü gibi dal listesinde aktif olan dal * işaretiyle gösterilmektedir. Yukarıda yer alan örnekte branch-a üzerinde çalıştığımız ve aktif olan daldır. Checkout komutunu ile başka bir dala geçtiğimiz andan itibaren, üzerinde çalıştığımız (working copy) dosyalar bu dalda yer alan dosyalarla değiştirilir. Yeni dal üzerinde yaptığımız tüm değişiklikler sadece bu dal bünyesinde kalacak şekilde Git tarafından yönetilir. Bu şekilde dallar arası izolasyonu sağlamak mümkün hale gelmektedir.

Checkout komutu kullanarak hem yeni bir dal oluşturabilir ve hem de akabinde bu dala geçişi şu şekilde yapabiliriz:

```
// Şema 2

// Branch-a dalına geçiş
[oacar@lin projem-klon]$ git checkout -b branch-a
Switched to a new branch 'branch-a'
```

```
// Dal listesi
[oacar@lin projem-klon]$ git branch
* branch-a
  master
[oacar@lin projem-klon]$
```

Şimdi branch-a üzerinde yeni bir dosya oluşturalım:

```
// Şema 3

// Yeni bir dosya oluşturulur
[oacar@lin projem-klon]$ touch dosya4

// Bu dosya indexe eklenir
[oacar@lin projem-klon]$ git add dosya4

// Commit oluşturulur
[oacar@lin projem-klon]$ git commit -m "dördüncü commit"
[branch-a e92f2b5] dördüncü commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya4

// Dal listesi
[oacar@lin projem-klon]$ git branch
* branch-a
  master

// Branch-a dalının commit geçmişi
[oacar@lin projem-klon]$ git log --oneline
e92f2b5 dördüncü commit
43750d9 üçüncü commit
a000068 ikinci commit
9c59180 ilk commit

// Dal bünyesinde bulunan ve üzerinde çalıştığımız dosyalar
[oacar@lin projem-klon]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya1
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya2
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya3
-rw-r----- 1 oacar users 0 Apr 22 15:10 dosya4
[oacar@lin projem-klon]$
```

Yeni dosyayı oluşturduktan sonra bu dosyayı commit komutuyla depoya ekledik. Branch-a

üzerinde yaptığımız en son commit e92f2b5 hash değerini taşımaktadır. Yaptığımız bu committen master dalının etkilenmediğini nasıl kontrol edebiliriz? Master dalına geçerek:

```
// Şema 4

// Master dalına geçiş
[oacar@lin projem-klon]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

// Master dalındaki dosyalar çalışma sahasına (working copy)
// aktarıldı.
[oacar@lin projem-klon]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya1
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya2
-rw-r----- 1 oacar users 0 Apr 22 14:54 dosya3

// Master dalının commit geçmişi
[oacar@lin projem-klon]$ git log --oneline
43750d9 üçüncü commit
a000068 ikinci commit
9c59180 ilk commit
[oacar@lin projem-klon]$
```

Görüldüğü gibi master dalında 9c59180, a000068 ve 43750d9 hash değerine sahip üç commit bulunmaktadır. Branch-a üzerinde oluşturduğumuz e92f2b5 hash değerli commit master dalına yansımamıştır. Bu master ve branch-a dallarının birbirlerinden bağımsız olduklarını göstermektedir.

Branch-a dalında iken çalışma dizininde dört dosyanın olduğunu görmekteyiz. Bu dosyalardan dosya4 isimli olanını yeni oluşturduk ve commit ile depoya ekledik. Checkout komutu ile master dalına geçtiğimizde, dosya4 isimli dosyanın çalıştığımız dizinde olmadığını görmekteyiz. Git dallar arası geçiş yapıldığında, dosyaları dalın içeriğine göre otomatik olarak değiştirmektedir. Bu şekilde gerçek anlamda dallar aracılığı ile birbirlerinden bağımsız görevler üzerinde paralel olarak çalışmak mümkün hale gelmektedir.

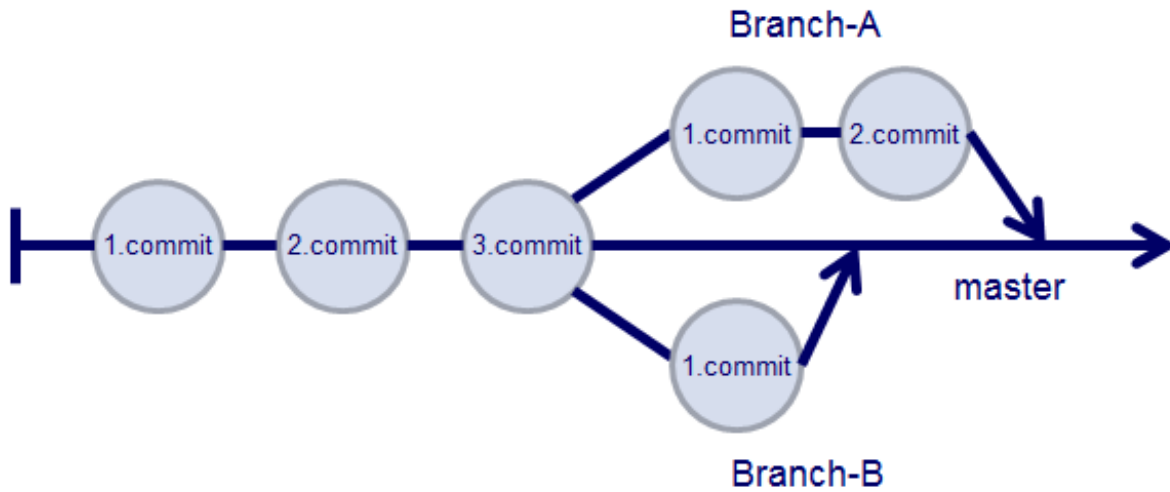
Benim günlük çalışmalarında dal konseptinin büyük bir yeri ve önemi var. Her yeni görev için yeni bir dal oluştururum. Git terminolojisinde bu dallara feature branch ismi verilmektedir. Bu şekilde birden fazla görev üzerinde birbirlerinden izole şekilde çalışmak mümkün hale gelmektedir. Aynı şekilde sürümlerde oluşan hatalar için dal konseptinden faydalanabiliriz. Bir sürüm hatasını gidermek ve yeni bir sürüm oluşturmak için bir bugfix-branch oluşturabiliriz. Bu dalın çıkış

noktası sürüm için seçilen commit olacaktır. Bu yeni dal üzerinde hatayı giderdikten sonra, bu daldan yeni bir sürüm oluşturabiliriz. Bu işlemleri yaparken kesinlikle ana geliştirme dalı olan master dalını rahatsız etmemiş oluyoruz. Tüm çalışmalarımız izole bir alanda gerçekleşmektedir.

Dalların Budanması

Ana geliştirme dalı olan master dalı projenin ömrü boyunca varlığını sürdürür. Kitabın ilerleyen bölümlerinde göreceğimiz gibi, master genelde sürüm oluşturmak için kullanılan ve yan dallarda meydana gelen değişikliklerin bir araya getirildiği ana daldır. Master dalını baz alarak oluşturduğumuz diğer dallar belli bir yaşam süresine sahiptirler. Bu yan dallarda yapılan değişikliklerin bir zaman sonra tekrar master dalına aktarılmaları yani budanmaları gerekmektedir. Aksi takdirde birbiriyle rekabet içinde olan ve aynı uygulamanın değişik versiyonlarını ihtiva eden karmaşık yapılar ortaya çıkabilir.

Bir dalın budama işlemi iki aşamadan oluşmaktadır. Birinci aşamada dal üzerinde yapılan değişiklikler master dalına aktarılır. İkinci aşamada dal silinir. Bu şekilde dal üzerinde yapılan tüm çalışmalar tekrar kaynağa yani master dalına geri dönmüş olur. Bunun ne anlama geldiğini resim 3 de görmekteyiz.



Resim 3

Budama işleminin ilk aşamasını gerçekleştirmek için git merge komutunu kullanmamız gerekiyor. Merge komutu ile bir daldaki dosyalar üzerinde yapılan değişiklikler diğer daldaki dosyalar üzerinde uygulanır. Bu şekilde dosyalar birleştirilmiş olur. Bunun nasıl yapıldığını şimdi bir örnek üzerinde inceleyelim.

Git merge işlemi için gerekli depoyu ve içeriğini şu şekilde oluşturabiliriz:

```
// Şema 5

rm -rf pratik-git
mkdir pratik-git
cd pratik-git
git init
touch dosya1 dosya2 dosya3
echo "test1" > dosya1
git add --all
git commit -m "dosya1 dosya2 dosya3 eklendi"
git checkout -b branch-a
touch dosya4 dosya5
git add --all
git commit -m "dosya4 dosya5 eklendi"
git checkout master
git log --oneline
git checkout branch-a
git log --oneline
git checkout master
git branch
```

Bu içeriği örneğin makerepo.sh ismindeki bir dosyada tutup, oluşturduğumuz bu bash scripti şu şekilde koşturabiliriz:

```
// Bash script dosyasını çalıştırılabilir hale getirir
[oacar@lin repo]$ chmod 755 makerepo.sh

// Bash scripti koşturur
[oacar@lin repo]$ ./makerepo.sh
```

Şimdi yeni oluşturduğumuz yerel depoyla çalışmaya başlayalım:

```
// Deponun bulunduğu dizine geçiyoruz
[oacar@lin repo]$ cd pratik-git/
```

Şema 5 de yer alan script tarafından depo bünyesinde oluşturulan branch-a dalına geçiyoruz:

```
[oacar@lin pratik-git]$ git checkout branch-a
Switched to branch 'branch-a'
```

Şimdi branch-a dalındayız.

```
[oacar@lin pratik-git]$ ls -l
total 4
```

```
-rw-r----- 1 oacar users 6 Apr 23 10:44 dosya1
-rw-r----- 1 oacar users 0 Apr 23 10:44 dosya2
-rw-r----- 1 oacar users 0 Apr 23 10:44 dosya3
[oacar@lin pratik-git]$
```

Şimdi dosya1 üzerinde bir takım değişiklikler yapalım ve bu değişiklikleri master ana dala aktaralım. Bu dosyanın içeriği aşağıda görülmektedir:

```
[oacar@lin pratik-git]$ cat dosya1
test1
[oacar@lin pratik-git]$
```

Bu dosyanın içeriğini şu şekilde değiştiriyoruz:

```
// Dosya1 isimli dosyaya ikinci bir satır ekliyoruz
[oacar@lin pratik-git]$ echo test2 >> dosya1

// Dosyanın yeni içeriği
[oacar@lin pratik-git]$ cat dosya1
test1
test2
[oacar@lin pratik-git]$
```

Dosya1 ismini taşıyan dosyaya test2 şeklinde yeni bir satır ekledik. Commit işlemini şu şekilde yapıyoruz:

```
// Şema 6

[oacar@lin pratik-git]$ git status
On branch branch-a
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   dosya1

no changes added to commit (use "git add" and/or "git commit -a")

// Dosya indexe eklenir
[oacar@lin pratik-git]$ git add dosya1

// Commit işlemi yapılır
[oacar@lin pratik-git]$ git commit -m "dosya1 e test2 satırı eklendi"
[branch-a 21932d6] dosya1 e test2 satırı eklendi
1 file changed, 1 insertion(+)
```

Branch-a dalının yeni commit listesi şu şekildedir:

```
[oacar@lin pratik-git]$ git log --oneline
21932d6 dosya1 e test2 satırı eklendi
879277e dosya4 dosya5 eklendi
d031c14 dosya1 dosya2 dosya3 eklendi
[oacar@lin pratik-git]$
```

Branch-a dalının gerekli tüm değişiklikleri ihtiva ettiğini ve artık budanması gerektiğini var sayalım. Bu durumda git merge komutu ile branch-a dalındaki değişiklikleri master dalına almamız gerekiyor. Merge işlemini şu şekilde gerçekleştirebiliriz:

```
// Şema 7

// Dal listesi
[oacar@lin pratik-git]$ git branch
* branch-a
  master
```

Master dalına geçiyoruz.

```
[oacar@lin pratik-git]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Merge işlemini gerçekleştiriyoruz.

```
[oacar@lin pratik-git]$ git merge branch-a
Updating d031c14..21932d6
Fast-forward
 dosya1 | 1 +
 dosya4 | 0
 dosya5 | 0
 3 files changed, 1 insertion(+)
 create mode 100644 dosya4
 create mode 100644 dosya5
[oacar@lin pratik-git]$
```

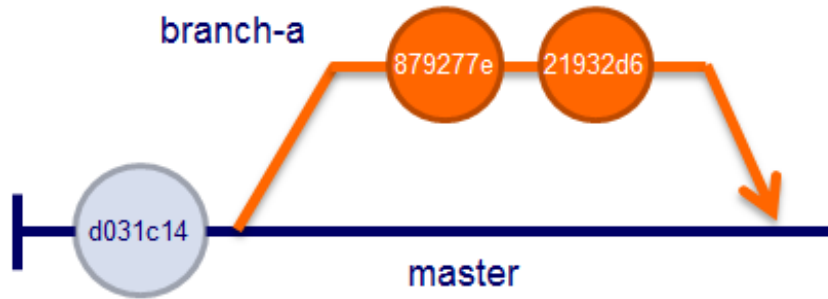
Merge işleminden sonra branch-a da dosya1 üzerinde yapılan değişiklikler şimdi master dalında yer alan dosya1 dosyasına eklendi.

```
[oacar@lin pratik-git]$ cat dosya1
```

```
test1
test2
[oacar@lin pratik-git]$
```

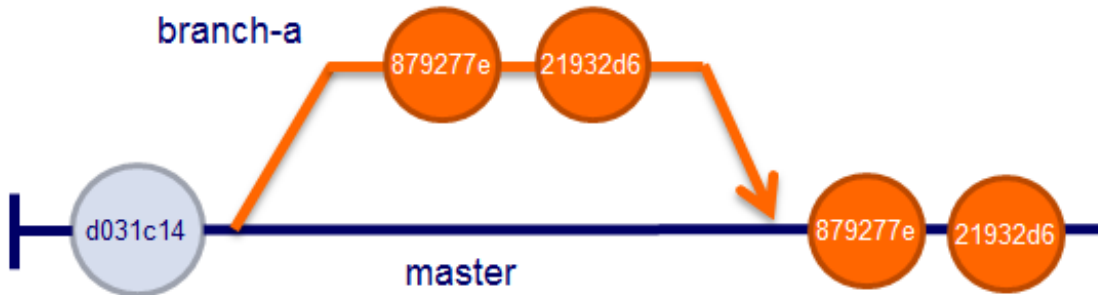
Merge işlemini gerçekleştirebilmek için değişikliklerin alınmak istendiği dala yani master dalına geçmemiz gerekiyor. Bunun için git checkout master komutunu kullanıyoruz. Akabinde git merge branch-a komutu ile branch-a dalında yaptığımız değişiklikleri master dalına alıyoruz. Merge işlemi esnasında ekranda Fast-forward çıktısını görmekteyiz. Bunun ne anlama geldiğini bir sonraki bölümde yakından inceleyeceğiz. Fast-forward kısaca branch-a dalında yer alan tüm commitlerin dosyalar üzerinde içeriksel bir çakışma olmadan master dalının commit listesine alındığı anlamına gelmektedir.

İki dalın birleştirilmesi işlemini şu şekilde görselleştirebiliriz. Resim 4 de merge işlemi öncesi durumu görmekteyiz.



Resim 4

Merge işleminden sonra master dalının yapısı resim 5 de yer almaktadır.



Resim 5

Branch-a isimli dalı sildikten sonra, master dalının yapısı resim 6 da gibi olacaktır.



Resim 6

Merge işlemi esnasında branch-a dalında yer alan 879277e ve 21932d6 hash değerlerindeki commitler fast-forward yöntemiyle master dalına birebir alınmışlardır. İki dalın birleştirildiğinin ibaresini master dalın commit listesinde görebiliriz:

```
[oacar@lin pratik-git]$ git log --oneline
21932d6 dosya1 e test2 satırı eklendi
879277e dosya4 dosya5 eklendi
d031c14 dosya1 dosya2 dosya3 eklendi
[oacar@lin pratik-git]$
```

Yaptığımız merge işlemi Git fast-forward olarak gerçekleştirdi, çünkü dosyalar birleştirilirken içeriksel bir çakışma yaşanmadı. Örneğin her iki dalda aynı dosya üzerinde değişiklik yapmış olsaydık, merge işlemi nasıl gerçekleşirdi? Şimdi bunu deneyelim.

Önce şema 5 de yer alan script ile yerel depomuzu oluşturuyoruz. Akabinde master dalında dosya1 isimli dosyayı şu şekilde değiştiriyoruz:

```
[oacar@lin repo]$ cd pratik-git/

// Master dalına geçiş
[oacar@lin repo]$ git checkout master

// Dosya1 isimli dosyanın içeriği
[oacar@lin pratik-git]$ cat dosya1
test1

// Dosya1 isimli dosyaya yeni satır eklenir
[oacar@lin pratik-git]$ echo "test3" >> dosya1

// Dosya1 dosyasının yeni içeriği
[oacar@lin pratik-git]$ cat dosya1
test1
test3

// Değişikliğe uğrayan dosya indexe eklenir
[oacar@lin pratik-git]$ git add dosya1
```

```
// Commit yapılır
[oacar@lin pratik-git]$ git commit -m "ikinci commit"
[master afbfalb] ikinci commit
1 file changed, 1 insertion(+)
[oacar@lin pratik-git]$
```

Master dalındaki dosya1 isimli dosyaya "test3" içeriğindeki yeni satırı ekledikten sonra, bu değişikliği depoya commit operasyonu ile ekliyoruz. Bu commit öncesinde branch-a isimli dalı oluşturduğumuz için master üzerinde yaptığımız değişiklikler branch-a isimli daha yansımamaktadır.

Şimdi branch-a dalına geçelim ve dosya1 dosyasını şu şekilde değiştirelim:

```
// Branch-a dalına geçiş
[oacar@lin pratik-git]$ git checkout branch-a
Switched to branch 'branch-a'

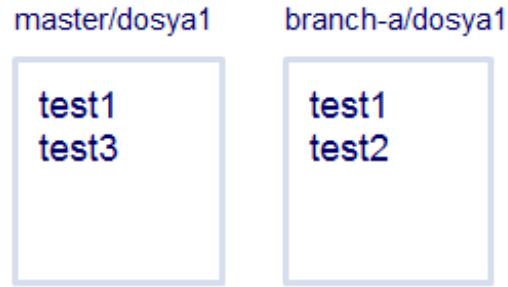
// Dosyanın içeriği
[oacar@lin pratik-git]$ cat dosya1
test1

// Yeni bir satır oluşturuldu
[oacar@lin pratik-git]$ echo "test2" >> dosya1

// Dosya indexe eklendi
[oacar@lin pratik-git]$ git add dosya1

// Commit yapıldı
[oacar@lin pratik-git]$ git commit -m "branch-a ikinci commit"
[branch-a b039be4] branch-a ikinci commit
1 file changed, 1 insertion(+)
[oacar@lin projem-klon]$
```

Paralel olarak iki dal üzerinde yaptığımız işlemleri şu şekilde özetleyebiliriz:



Resim 7

Eğer branch-a dalındaki dosya1 değişikliğini master dalına almak istersek, bir sorun çıkacağını görmekteyiz, çünkü aynı dosyanın ikinci satırı her iki dalda başka bir değere sahip. Hangi değişiklik doğru olanıdır? Bu sorunun cevabını her zaman veremeyebiliriz, lakin merge işlemi esnasında bir hatanın yani bir içeriksel çakışmanın doğacağı ortada. Bu gibi durumlara merge conflict ismi verilmektedir. Merge conflict oluştuğunda, yani merge esnasında birbirlerine içeriksel aykırı dosyalar birleştirildiğinde oluşan hataları elden düzeltmemiz gerekiyor. Öncelikle merge conflict in nasıl oluştuğunu inceleyelim. Bu amaçla master dalı ile branch-a dalını birleştiriyoruz yani branch-a dalını buduyoruz.

```
// Master dalına geçiş
[oacar@lin pratik-git]$ git checkout master
Switched to branch 'master'

// Merge işlemi
[oacar@lin pratik-git]$ git merge branch-a
Auto-merging dosya1
CONFLICT (content): Merge conflict in dosya1
Automatic merge failed; fix conflicts and then commit the result.
[oacar@lin pratik-git]$
```

"Merge conflict in dosya1" hatasını aldık. Git tarafından iki dosya birleştirilmek istendi, lakin ikinci satır her iki dosyada da başka bir değere sahip olduğundan, birleştirme işlemi başarıyla tamamlanamadı. Şimdi duruma müdahale etmemiz ve hatayı gidermemiz gerekiyor. Öncelikle nasıl bir çakışmanın (conflict) olduğunu anlamak için dosya1 dosyasının içeriğine bir göz atalım:

```
[oacar@lin pratik-git]]$ cat dosya1
test1
<<<<<<< HEAD
test3
=====
test2
```



```
>>>>>> branch-a
[oacar@lin pratik-git]]$
```

Git merge işlemi esnasında iki dosyanın değişik olan yanlarını hatayı gidermemiz için dosya1 dosyasında bir araya getirmiştir. Burada gördüğümüz şey şudur:

```
<<<<<<< HEAD
test3
```

Bu master dalında yer alan dosya1 isimli dosyanın ikinci satırında yapılan değişikliği göstermektedir.

```
=====
```

Bu işaret zinciri iki değişiklik arasındaki farklılığı göstermek için kullanılan ayrıştırma çizgisidir.

```
test2
>>>>>> branch-a
```

Bu branch-a bünyesindeki dosya1 dosyanın ikinci satırında yapılan değişikliği ihtiva etmektedir.

Burada kullanıcı olarak hatayı (merge conflict) gidermek bize düşmektedir. Hatanın ne olduğunu tam olarak görmek için git status komutunu şema 8 de görüldüğü gibi kullanabiliriz.

```
// Şema 8

[oacar@lin pratik-git]$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Changes to be committed:

    new file:   dosya4
    new file:   dosya5

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

    both modified:   dosya1

[oacar@lin pratik-git]$
```

Ekran çıktısının "Unmerged paths" bölümünde merge conflict oluşan dosyaları görmekteyiz. Aynı şekilde git diff komutu ile iki dosya arasındaki farklılığı ekranda görebiliriz. Bu arada "Changes to be committed" bölümünde dosya4 ve dosya5 yeni dosya olarak gösterilmekte. Bu iki dosyayı merge esnasında branch-a dan aldık.

```
[oacar@lin pratik-git]$ git diff
diff --cc dosya1
index cd85853,bae42c5..0000000
--- a/dosya1
+++ b/dosya1
@@@ -1,2 -1,2 +1,6 @@@
     test1
++<<<<<<< HEAD
     +test3
++=====
+ test2
++>>>>>>> branch-a
[oacar@lin pratik-git]$
```

Git diff çıktısını şu şekilde açıklayabiliriz:

```
--- a/dosya1
+++ b/dosya1
```

Söz konusu olan dosyanın ismi dosya1 dir.

```
++<<<<<<< HEAD
     +test3
```

Master/dosya1 dosyasına test3 değerindeki satır eklenmiştir.

```
+ test2
++>>>>>>> branch-a
```

Branch-a/dosya1 dosyasına test2 değerindeki satır eklemiştir.

Oluşan bu hatayı doğrudan dosya1 isimli dosya üzerinde giderebiliriz. Bunun yanı sıra git mergetool komutu editörlerin gerekli içerikle açılarak, hatayı gidermemizi kolaylaştırmaktadır. Git mergetool komutu şu şekilde kullanabiliriz:

```
// Şema 9

[oacar@lin pratik-git]$ git mergetool
```

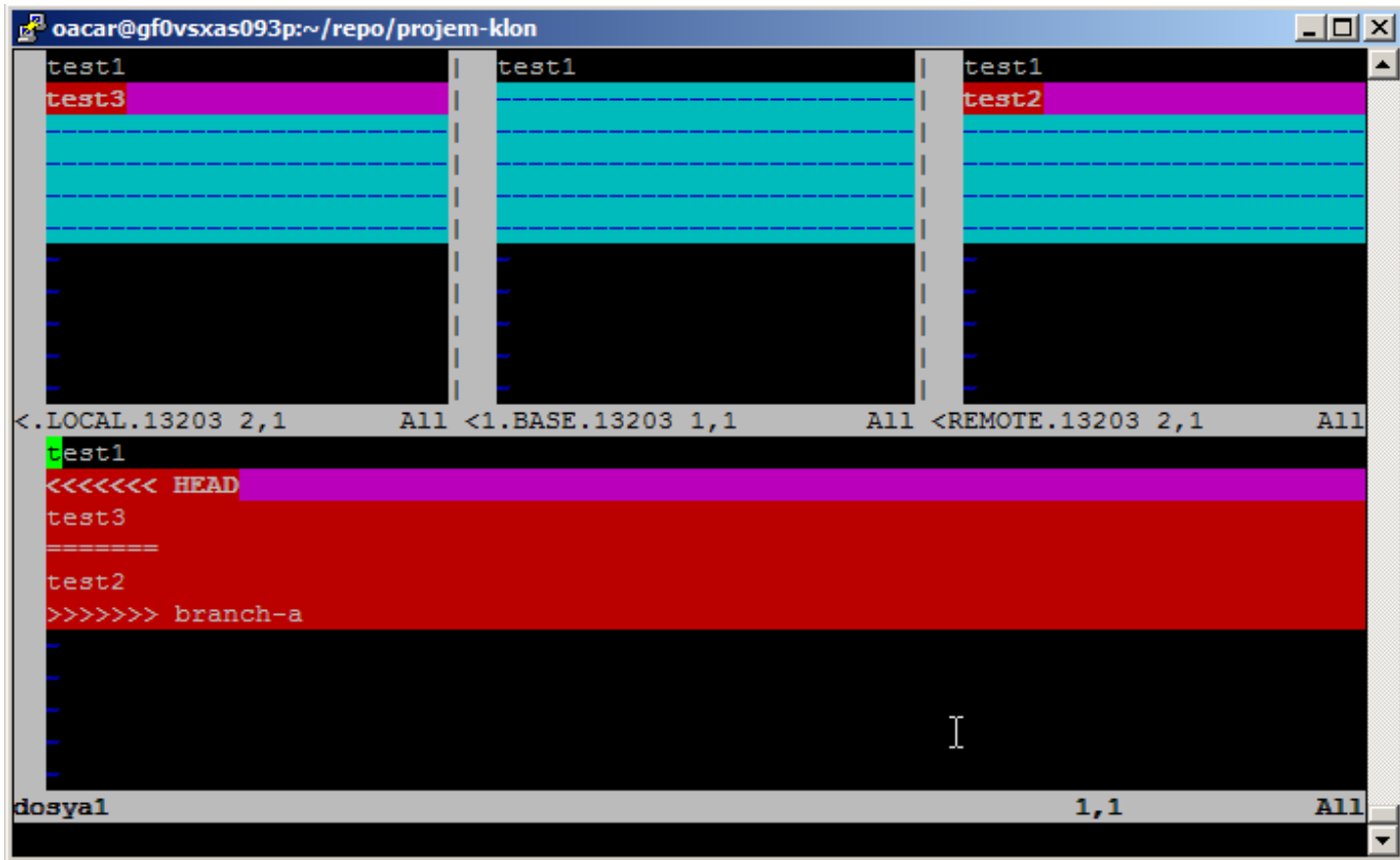
```

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
tortoisemerge emerge vimdiff
Merging:
dosyal

Normal merge conflict for 'dosyal':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):

```

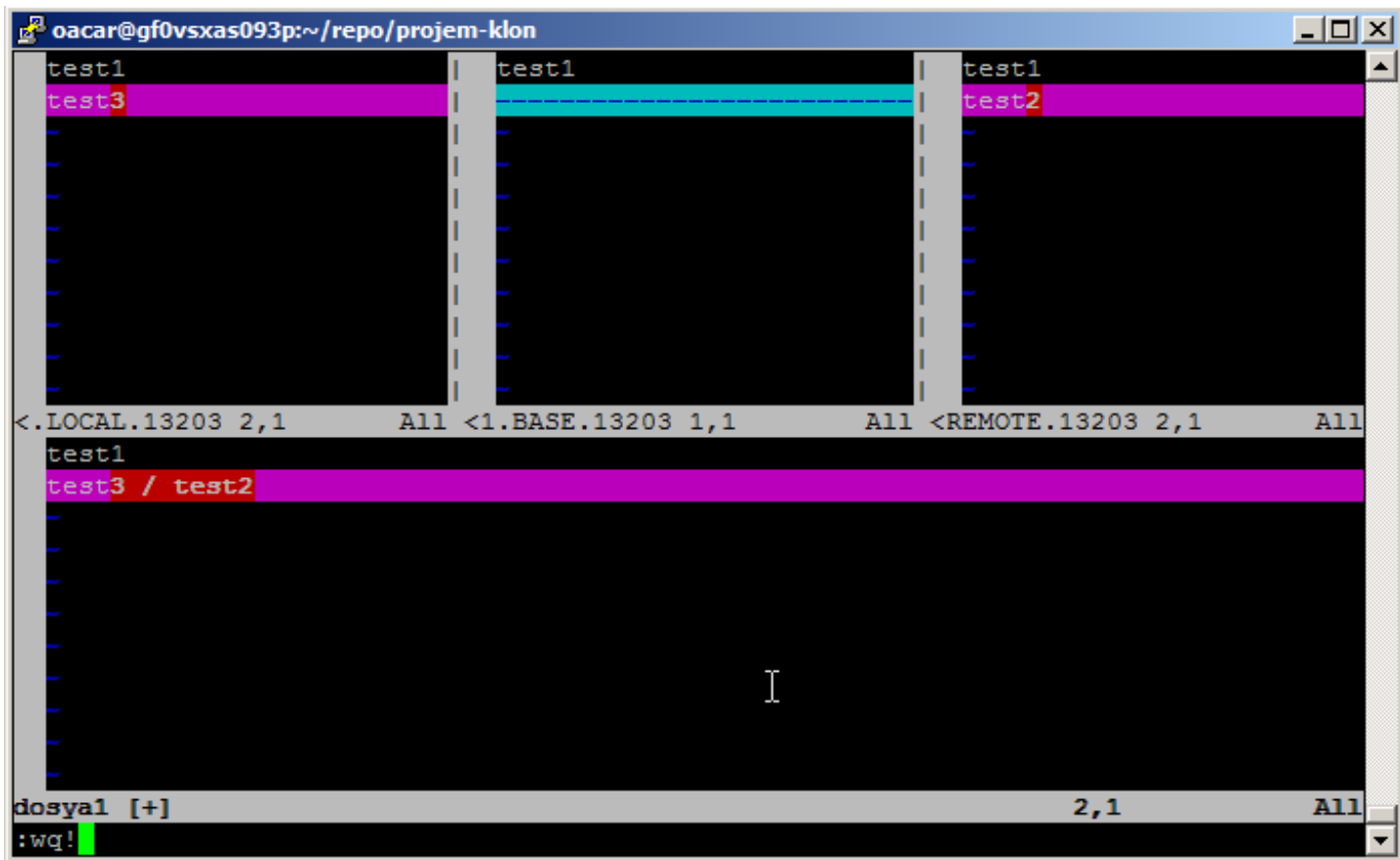
Kullanmak istediğimiz komutu git config dosyasında "merge.tool" anahtarı ile tanımlayabiliriz. Şema 9 da bir editörün tanımlı olmadığını görüyoruz, bu yüzden Git vi editörünü kullanmayı teklif ediyor.



Resim 8

Resim 8 de görüldüğü gibi Git dört vi editörü açtı. En soldaki editörde master/dosya1, ortada bir commit öncesindeki dosya1 isimli dosyanın içeriği, sağdaki editörde branch-a/dosya1 yer almaktadır. Altteki editörde iki dosyanın birleştirilmiş halini görmekteyiz. Bu editörde oluşan

çakışmayı ortadan kaldırabiliriz. Ben dosya1 in içeriğini şu şekilde değiştiriyorum.



Resim 9

Dosya1 in içeriği şimdi şu şekilde oldu:

```
[oacar@lin pratik-git]$ cat dosya1
test1
test3 / test2
[oacar@lin pratik-git]$
```

Git'e göre dosya1 isimli dosya üzerinde değişiklik yaptığımız için bu değişikliği add ve commit komutları ile depoya tanıtmamız gerekiyor. Önce git status ile hangi durumda olduğumuzu inceliyoruz:

```
[oacar@lin pratik-git]$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
```

```
modified:   dosya1
new file:   dosya4
new file:   dosya5
```

```
[oacar@lin pratik-git]$
```

Şimdi add ve commit ile bu değişikliği depoya tanıtabiliriz:

```
// Dosya indexe eklenir
[oacar@lin pratik-git]$ git add --all

// Commit işlemi
[oacar@lin pratik-git]$ git commit -m "dosya1 merge işlemi"
[master c10ddf6] dosya1 merge işlemi
[oacar@lin pratik-git]$
```

Şimdi commit listesine bir göz atalım:

```
[oacar@lin pratik-git]$ git log --oneline
c10ddf6 dosya1 merge işlemi
b039be4 branch-a ikinci commit
afbfa1b ikinci commit
b8a823c dosya4 dosya5 eklendi
d2a9c38 dosya1 dosya2 dosya3 eklendi
[oacar@lin pratik-git]$
```

Commit listesindeki b039be4 hash değerindeki commitin branch-a da dosya1 üzerinde yapılan işlem ardından oluşturulan commit olduğunu görmekteyiz. Hemen branch-a dalının commit geçmişine bir göz atalım:

```
[oacar@lin pratik-git]$ git log branch-a --oneline
b039be4 branch-a ikinci commit
b8a823c dosya4 dosya5 eklendi
d2a9c38 dosya1 dosya2 dosya3 eklendi
```

Git merge işlemi esnasında daha önceki örneklerde de gördüğümüz gibi merge edilen dalın tüm commit tarihçesini master dalına almıştır. Bunun yanı sıra oluşan merge conflicti yok etmek için c10ddf6 hash değerinde yeni bir commit oluştu. Oluşan bir commitden ötürü bu merge işlemine 3-way merge ismi verilmektedir. Git her iki commiti baz alarak, yeni bir commit oluşturmaktadır. Bunu oluşan son commitin içeriğine baktığımızda, görmekteyiz:

```
[oacar@lin pratik-git]$ git show c10ddf6
commit c10ddf63ec3bedef877135d9108ec8c9c044c8b3
```

```

Merge: afbfa1b b039be4
Author: Özcan <acar@agilementor.com>
Date:   Thu Apr 30 16:00:25 2015 +0200

```

dosya1 merge işlemi

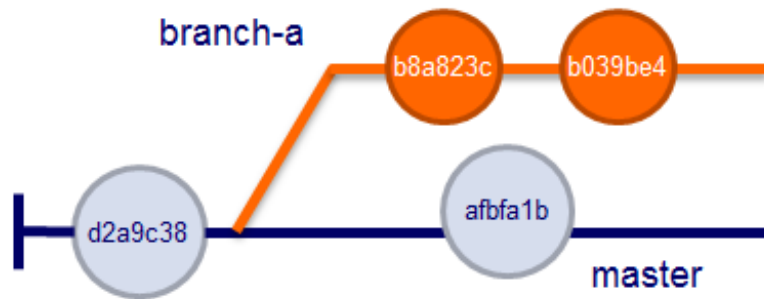
```

diff --cc dosya1
index cd85853,bae42c5..e1c1682
--- a/dosya1
+++ b/dosya1
@@@ -1,2 -1,2 +1,2 @@@
  test1
- test3
- test2
++test3 / test2
[oacar@lin pratik-git]$

```

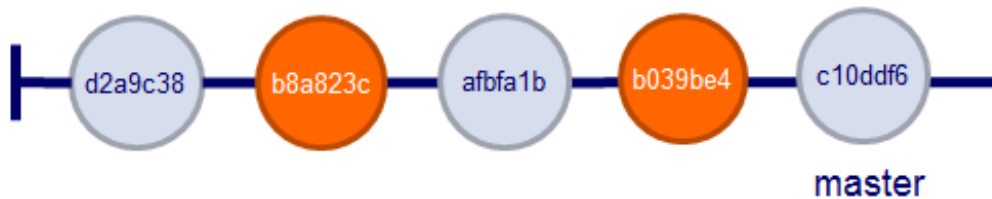
"Merge: afbfa1b b039be4" mesajı son merge commit için iki commitin kullanıldığını göstermektedir. Bu oluşan commitin iki başı (parent) olduğu anlamına gelmektedir. Bu başlardan bir tanesi master bünyesi yer alan afbfa1b hash değerindeki, diğeri branch-a dalının b039be4 hash değerindeki commitidir.

Merge öncesindeki yapıyı şu şekilde görselleştirebiliriz:



Resim 10

Merge sonrasındaki master dal yapısı şu şekildedir:



Resim 11

Git log --graph komutunu kullanarak, dalların birbirleriyle olan ilişkisinin ekran çıktısını şu şekilde alabiliriz:

```
// Şema 10

[oacar@lin pratik-git]$ git log --graph
*   commit c10ddf66d1ddd85408eeb8dcd9363eb3c6294f0f
|\  Merge: 824b72a 548f639
| | Author: Özcan <acar@agilementor.com>
| | Date:   Thu Apr 30 16:22:27 2015 +0200
| |
| |     dosya1 merge işlemi
| |
| * commit b039be47f8ae7a9b1029e442545aa3a655b0a9d
| | Author: Özcan <acar@agilementor.com>
| | Date:   Thu Apr 30 16:10:40 2015 +0200
| |
| |     branch-a ikinci commit
| |
| * commit b8a823c4b01d09ce70d69f30af415127a6b776ce
| | Author: Özcan <acar@agilementor.com>
| | Date:   Thu Apr 30 16:09:34 2015 +0200
| |
| |     dosya4 dosya5 eklendi
| |
* | commit afbfa1b6ab5d178308b5cd35e27d44dd8f65361
|/  Author: Özcan <acar@agilementor.com>
|   Date:   Thu Apr 30 16:10:11 2015 +0200
|
|       ikinci commit
|
* commit d2a9c38596b17b2c3c3dc7b68d2f103d6be8297
   Author: Özcan <acar@agilementor.com>
   Date:   Thu Apr 30 16:09:34 2015 +0200

       dosya1 dosya2 dosya3 eklendi
[oacar@lin projem-klon]$
```

Şema 10 da yer alan ekran çıktısı master dalının commit tarihçesini göstermektedir. Bu ekran çıktısında merge işlemi sonrasında master ve branch-a üzerinde yapılan tüm işlemleri görmekteyiz. Buna göre master dalındaki ilk commit sonrasında (d2a9c38) branch-a dalını oluşturduk. Branch-a dalı üzerinde b8a823c değerindeki ilk commiti gerçekleştirdik. Buna paralel olarak master dalında afbfa1b değerindeki ikinci commiti oluşturduk. Bu master dalındaki dosya1 üzerinde yaptığımız değişikliği yansıtmaktadır. Daha sonra branch-a dalında dosya1 üzerindeki değişikliği (b039be4)

gerçekleştirdik. Akabinde merge komutu ile iki dalı birleştirdik. Bu birleşmeden c10ddf6 hash değerindeki commit doğdu. Bu commit ile iki dal birleştirilmiş oldu. Bu noktadan itibaren branch-a dalını silebiliriz.

Merge işlemini kendimiz koordine etmek zorunda kaldık. Bunun sebebi her iki dalda dosya1 üzerinde değişiklik yapıldığı için fast-forward merge işleminin gerçekleşmemiş olmasıdır. Şimdi fast-forward merge işlemi yakından inceleyelim.

Git Fast-Forward Merge

Bir önceki bölümde master ve branch-a dallarını birleştirirken, fast-forward mesajını almıştık. Bunun ne anlama geldiğini bu bölümde tekrar inceleyelim. Bu bölümde örnek olan depoyu oluşturmak için şema 5 deki scripti kullandım.

Bu scripti kullanarak, pratik-git ismini taşıyan bir depo oluşturuyoruz. Bu depo bünyesinde iki dal bulunuyor: master ve branch-a. Bu dalların sahip olduğu commit geçmişi şu şekildedir:

```
// Master dalına geçiş
[oacar@lin pratik-git]$ git checkout master
Already on 'master'

// Bu dalın geçmişi
[oacar@lin pratik-git]$ git log --oneline
73be77a dosya1 dosya2 dosya3 eklendi

// Branch-a dalına geçiş
[oacar@lin pratik-git]$ git checkout branch-a
Switched to branch 'branch-a'

// Bu dalın geçmişi
[oacar@lin pratik-git]$ git log --oneline
a2a4725 dosya4 dosya5 eklendi
73be77a dosya1 dosya2 dosya3 eklendi
[oacar@lin pratik-git]$
```

Branch-a dalı master dalına göre bir commit ileridedir. Şimdi branch-a dalında yeni bir commit oluşturup, branch-a dalının sahip olduğu tüm değişiklikleri master dalına çekelim:

```
// Branch-a dalındayız
[oacar@lin pratik-git]$ git checkout branch-a

// Bu dalın dizin yapısı
```



```
[oacar@lin pratik-git]$ ls -l
total 4
-rw-r----- 1 oacar users 6 Apr 30 16:46 dosya1
-rw-r----- 1 oacar users 0 Apr 30 16:46 dosya2
-rw-r----- 1 oacar users 0 Apr 30 16:46 dosya3
-rw-r----- 1 oacar users 0 Apr 30 16:49 dosya4
-rw-r----- 1 oacar users 0 Apr 30 16:49 dosya5

// Yeni bir dosya oluşturduk
[oacar@lin pratik-git]$ touch dosya6

// Dosya indexe eklendi
[oacar@lin pratik-git]$ git add dosya6

// Commit yapıldı
[oacar@lin pratik-git]$ git commit -m "dosya6 eklendi"
[branch-a 1ca187f] dosya6 eklendi
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dosya6
[oacar@lin pratik-git]$
```

Şimdi merge işlemini gerçekleştirmek için master dalına dönüyoruz:

```
// Master dalına geçiş
[oacar@lin pratik-git]$ git checkout master
Switched to branch 'master'

// Merge işlemi
[oacar@lin pratik-git]$ git merge branch-a
Updating 73be77a..1ca187f
Fast-forward
 dosya4 | 0
 dosya5 | 0
 dosya6 | 0
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dosya4
 create mode 100644 dosya5
 create mode 100644 dosya6
[oacar@lin pratik-git]$
```

Şimdi master dalının commit geçmişine bir göz atalım:

```
[oacar@lin pratik-git]$ git log --graph
* commit 1ca187f1be1afd40ff28e5630d96aa48160270b
| Author: Özcan <acar@agilementor.com>
| Date: Thu Apr 30 16:50:35 2015 +0200
```

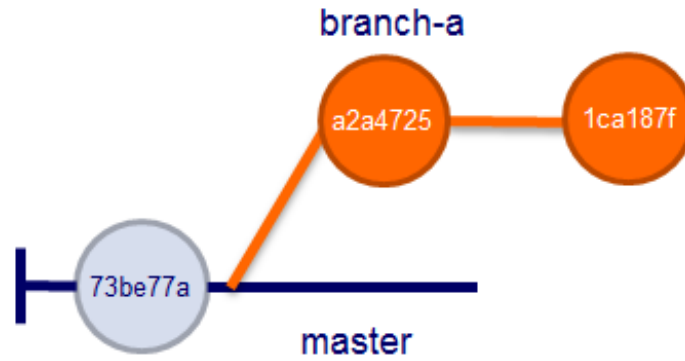
```

|
|   dosya6 eklendi
|
* commit a2a4725af954386ef48c7e546193f960d25336c0
| Author: Özcan <acar@agilementor.com>
| Date:   Thu Apr 30 16:46:53 2015 +0200
|
|   dosya4 dosya5 eklendi
|
* commit 73be77a406a56b6a10123415e32e76f46097808d
  Author: Özcan <acar@agilementor.com>
  Date:   Thu Apr 30 16:46:53 2015 +0200

    dosya1 dosya2 dosya3 eklendi
[oacar@lin pratik-git]$

```

Görüldüğü gibi bir fast-forward merge oluştu. Git, branch-a dalındaki a2a4725 ve 1ca187f hash değerlerindeki commitleri alıp, master dalının en tepesindeki commitin üzerine ekledi. Eğer branch-a bünyesinde beş değişik commit yapmış olsaydık, fast-forward merge sonucu bu beş commiti master dalındaki 73be77a hash değerindeki son commitin üzerine eklendiğini görürdük. Eğer master dalı commit olarak ileri gitmediyse, yani branch-a dalının başlangıcı hala güncel olan master dalının en son commitine eşitse, bu durumda merge işlemi sırasında tüm commitler bir daldan, diğer dala birebir aktarılmaktadırlar. Bu işleme fast-forward (hızlı ileri) ismi verilmektedir.



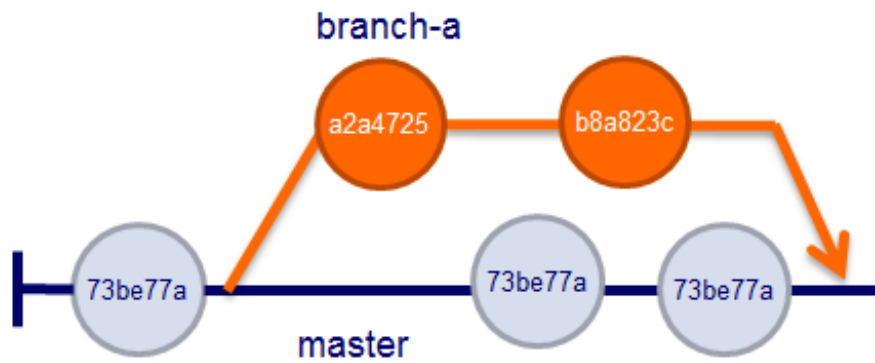
Resim 11.1

Resim 11.1 fast-forward merge öncesi master ve branch-a dallarının içeriğini göstermektedir. Merge sonrasında branch-a dalında yer alan commitlerin birebir master dalına aktarıldıklarını görmekteyiz.



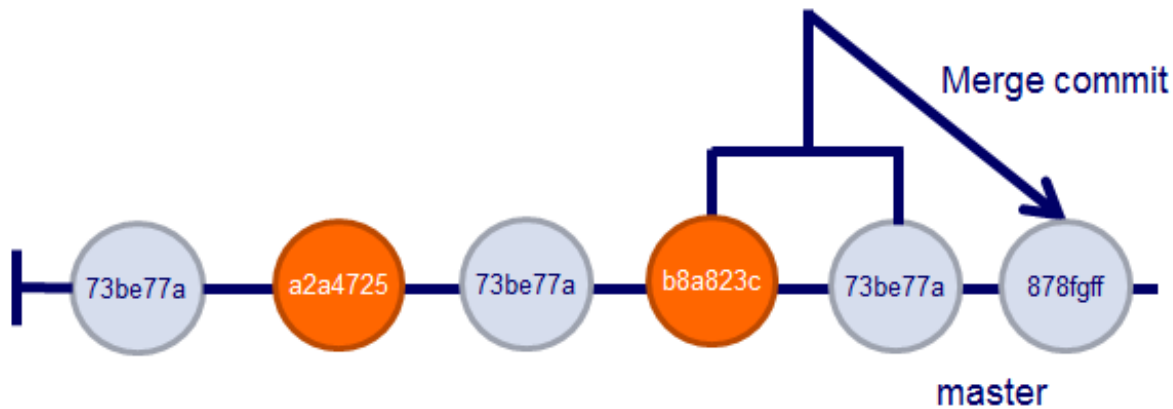
Resim 11.2

Peki ne zaman fast-forward merge oluşmaz? Resim 11.3 de görüldüğü gibi master alıp başını giderse, fast-forward merge yapmak mümkün olmaz. Bu durumda branch-a dalında yapılan değişiklikler için master bünyesinde merge commit oluşturulması gerekmektedir.



Resim 11.3

Yapılan merge commiti resim 11.4 de görmekteyiz.



Resim 11.4

Deponun Tarihçesi

Reflog komutu ile yerel bir deponun tarihçesini yani dal bünyesinde olup bitenlerin listesini şu şekilde edinebiliriz:

```
// Şema 11

[oacar@lin projem-klon]$ git reflog
98d5e7a HEAD@{0}: checkout: moving from branch-a to master
00cc582 HEAD@{1}: checkout: moving from master to branch-a
98d5e7a HEAD@{2}: commit (merge): dosyal merge işlemi
2d82c1b HEAD@{3}: checkout: moving from branch-a to master
00cc582 HEAD@{4}: commit: branch-a ikinci commit
fa95c0f HEAD@{5}: checkout: moving from master to branch-a
2d82c1b HEAD@{6}: commit: ikinci commit
fa95c0f HEAD@{7}: clone: from /export/home/oacar/repo/projem.git/
```

Bu listeyi aşağıdan yukarıya doğru okumamız gerekmektedir. Depo bünyesinde yapılan en son işlemler listenin başında yer almaktadır.

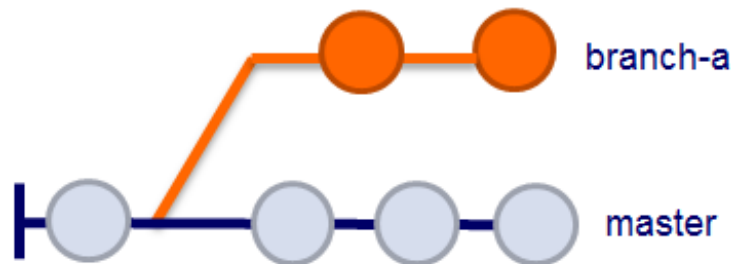
Reset komutu yardımı ile depo tarihçesindeki herhangi bir commite geri dönebiliriz:

```
$ git reset --hard HEAD@{4}
```

Reset üzerinde çalıştığımız dosyaları 00cc582 commitinde sahip oldukları duruma geri çevirecektir.

Git Rebase

İki dalı birleştirmenin diğer bir yolu rebase işlemidir. Rebase işlemini açıklamak için resim 12 de yer alan içeriğe bakalım.

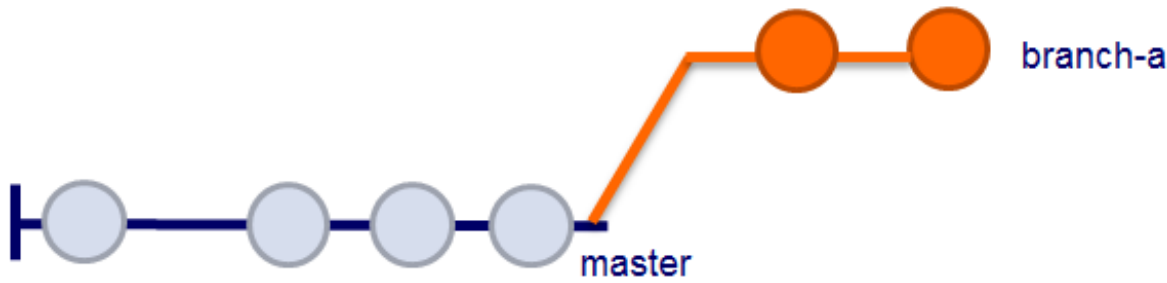


Resim 12

Resim 12 de master ve branch-a ismini taşıyan iki dal görmekteyiz. Branch-a master dalındaki ilk

commit işleminden sonra oluşturulmuş ve kendi bünyesinde iki commit taşımaktadır. Branch-a oluşturulduktan sonra master bünyesinde üç commit gerçekleşmiş. Bu şekilde master ve branch-a dallarının arası içerik olarak açılmış görünüyor. Merge işlemini yaparak, iki dalı bir araya getirebiliriz. Bu bölümde rebase mekanizmasını kullanarak, iki dalın nasıl birleştirilebileceğini göstermek istiyorum.

Rebase genelde yeni bir dalın, master üzerinde değişikliklerin oluşması ve yeni dalın başlangıcının bu değişiklikleri temel alacak şekilde değiştirilmesi gerektiğinde yapılan işlemidir. Örneğin resim 12 ye baktığımızda, branch-a dalını oluşturduktan sonra master dalında üç commitin oluştuğunu görmekteyiz. Eğer branch-a dalını master üzerindeki ilk commit değil de, dördüncü commitden sonra oluştursak daha iyi olurdu diyorsak, o zaman branch-a dalını master dalındaki en son commiti temel alacak şekilde yeniden yapılandırmamız gerekmektedir. Bunun için branch-a dalını silmek zorunda değiliz. Bu branch-a üzerinde yaptığımız iki commiti kaybetmemiz anlamına gelmektedir. Merge işlemi yapmamız da yerinde olmayabilir, çünkü master üzerindeki değişiklikleri kendi değişikliklerimizle birleştirmek istemeyebiliriz. Bu durumda rebase işlemini uygulayabiliriz. Rebase mekanizması master ve branch-a dalını birleştirerek, branch-a dalının güncellenmesini ve başlangıcının tayin ettiğimiz master commit seviyesine getirilmesini sağlamaktadır. Bunun nasıl yapıldığını resim 13 de görmekteyiz.



Resim 13

Branch-a bünyesinde master dalını baz alan rebase işlemini gerçekleştirdiğimizde, Git branch-a üzerinde yaptığımız commitleri bir kenara almakta ve master üzerinde yapılmış olan commitleri branch-a üzerinde uygulamaktadır. Bu bir fast-forward merge işlemidir, çünkü branch-a üzerinde yaptığımız değişiklikleri bir kenara aldığımızda, branch-a master ile birinci commit seviyesinde aynı yapıdadır. Bu durumda Git master üzerindeki birinci commitden sonra oluşan commitleri tek, tek branch-a dalına aktarır ve akabinde bir kenara koyduğu branch-a commitlerini branch-a dalının son hali üzerinde uygular. Bu sayede branch-a dalı master dalının dördüncü commitini temel alacak şekilde yeniden yapılandırılmış olur.

Şimdi rebase işleminin nasıl yapıldığı bir örnek üzerinde inceleyelim. Üzerinde çalıştığımız

deponun commit listesi aşağıda yer almaktadır:

```
[oacar@lin projem2]$ git log --oneline
af599b1 ilk commit: dosya1 eklendi
[oacar@lin projem2]$
```

Şimdi yeni bir dal oluşturuyoruz:

```
// Master dalını temel alan branch-a isimli dalı
// oluşturur
[oacar@lin projem2]$ git branch branch-a master

// Mevcut dal listesi
[oacar@lin projem2]$ git branch
  branch-a
* master
[oacar@lin projem2]$
```

Yeni dal üzerinde iki commit oluşturuyoruz.

```
// Branch-a dalına geçiş
[oacar@lin projem2]$ git checkout branch-a

// Metin1 isimli yeni bir dosya oluşturur
[oacar@lin projem2]$ touch metin1

// Bu dosya indexe eklenir
[oacar@lin projem2]$ git add --all

// Commit yapılır
[oacar@lin projem2]$ git commit -m "branch-a birinci commit: metin1 eklendi"
[branch-a c15ecce] branch-a birinci commit: metin1 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 metin1

// Metin2 isimli yeni bir dosya oluşturur
[oacar@lin projem2]$ touch metin2

// Bu dosya indexe eklenir
[oacar@lin projem2]$ git add --all

// Commit yapılır
[oacar@lin projem2]$ git commit -m "branch-a ikinci commit: metin2 eklendi"
[branch-a 4d9db4f] branch-a ikinci commit: metin2 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 metin2
[oacar@lin projem2]$
```

Master bünyesinde üç yeni commit oluşturalım:

```
// Master dalına geçiş
[oacar@lin projem2]$ git checkout master

// Dosya2 isimli yeni bir dosya oluşturur
[oacar@lin projem2]$ touch dosya2

// Bu dosya indexe eklenir
[oacar@lin projem2]$ git add --all

// Commit yapılır
[oacar@lin projem2]$ git commit -m "ikinci commit: dosya2 eklendi"
[master 7508a51] ikinci commit: dosya2 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya2

// Dosya3 isimli yeni bir dosya oluşturur
[oacar@lin projem2]$ touch dosya3

// Bu dosya indexe eklenir
[oacar@lin projem2]$ git add --all

// Commit yapılır
[oacar@lin projem2]$ git commit -m "üçüncü commit: dosya3 eklendi"
[master b803a7f] üçüncü commit: dosya3 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya3

// Dosya4 isimli yeni bir dosya oluşturur
[oacar@lin projem2]$ touch dosya4

// Bu dosya indexe eklenir
[oacar@lin projem2]$ git add --all

// Commit yapılır
[oacar@lin projem2]$ git commit -m "dördüncü commit: dosya4 eklendi"
[master d77d8e4] dördüncü commit: dosya4 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya4
[oacar@lin projem2]$
```

Şimdi master üzerindeki commit listesine bir göz atalım:

```
[oacar@lin projem2]$ git log --oneline
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
af599b1 dosya1 eklendi
[oacar@lin projem2]$
```

Dizin yapısı şöyledir:

```
[oacar@lin projem2]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 24 13:22 dosya1
-rw-r----- 1 oacar users 0 Apr 24 13:26 dosya2
-rw-r----- 1 oacar users 0 Apr 24 13:26 dosya3
-rw-r----- 1 oacar users 0 Apr 24 13:26 dosya4
[oacar@lin projem2]$
```

Şimdi branch-a dalına bir göz atalım:

```
// Branch-a dalına geçiş
[oacar@lin projem2]$ git checkout branch-a

// Dalın tarihçesi
[oacar@lin projem2]$ git log --oneline
8b7ce7d branch-a ikinci commit: metin2 eklendi
7b600fb branch-a birinci commit: metin1 eklendi
af599b1 ilk commit: dosya1 eklendi
[oacar@lin projem2]$
```

Böylece resim 12 deki yapıyı oluşturmuş olduk. Eğer bu noktada master bünyesinde yer alan değişiklikleri merge ile branch-a dalına alırsak, şöyle bir görüntü oluşacaktır:

```
// Branch-a dalına geçiş
[oacar@lin projem2]$ git checkout branch-a

// Merge işlemi
[oacar@lin projem2]$ git merge branch-a master
Merge made by the 'recursive' strategy.
 dosya2 | 0
 dosya3 | 0
 dosya4 | 0
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dosya2
create mode 100644 dosya3
```



```

create mode 100644 dosya4
[oacar@lin projem2]$

// Dalın tarihçesi
[oacar@lin projem2]$ git log --oneline
ae17427 Merge branch 'master' into branch-a
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
8b7ce7d branch-a ikinci commit: metin2 eklendi
7b600fb branch-a birinci commit: metin1 eklendi
af599b1 ilk commit: dosya1 eklendi

```

Merge ile master bünyesindeki tüm değişiklikleri branch-a alına çektik, lakin commit listesine baktığımızda, ae17427 hash değerine sahip bir bir merge-commitin oluştuğunu görmekteyiz. Bunun yanı sıra master bünyesindeki 2d0ce05, babaf11 ve 054671e hash değerindeki commitler branch-a üzerinde yer alan commitlerden sonra gelmektedirler. Burada merge işleminin uygun olmadığını görmekteyiz.

```

[oacar@lin projem2-bak]$ git checkout branch-a
[oacar@lin projem2-bak]$ git log --graph
*   commit ae17427ae845d1b141a0dfa257d5e81bd9a8410b
|\  Merge: 8b7ce7d 054671e
| | Author: Özcan Acar <oezcanacar@kpmg.com>
| | Date:   Thu Aug 13 11:21:33 2015 +0200
| |
| |     Merge branch 'master' into branch-a
| |
| * commit 054671e561325e125c643ace6d73413f0e63b5be
| | Author: Özcan Acar <oezcanacar@kpmg.com>
| | Date:   Thu Aug 13 11:19:52 2015 +0200
| |
| |     dördüncü commit: dosya4 eklendi
| |
| * commit babaf11f65a533288176ffabc3991b6733f22ec0
| | Author: Özcan Acar <oezcanacar@kpmg.com>
| | Date:   Thu Aug 13 11:19:05 2015 +0200
| |
| |     üçüncü commit: dosya3 eklendi
| |
| * commit 2d0ce05a41727754609e53e12b89d17c6f080b7f
| | Author: Özcan Acar <oezcanacar@kpmg.com>
| | Date:   Thu Aug 13 11:18:49 2015 +0200
| |
| |     ikinci commit: dosya2 eklendi

```

```
| |
* | commit 8b7ce7d1dfa9d9ef95fd753a15fcd43516d61a29
| | Author: Özcan Acar <oezcanacar@kpmg.com>
| | Date: Thu Aug 13 11:18:23 2015 +0200
| |
| |     branch-a ikinci commit: metin2 eklendi
| |
* | commit 7b600fb40fd0a2e5fefccce3683f78369c54eade
|/ Author: Özcan Acar <oezcanacar@kpmg.com>
| Date: Thu Aug 13 11:17:50 2015 +0200
|
|     branch-a birinci commit: metin1 eklendi
|
* commit af599b18088eda3c01e1856329daaaea2ecbdfef0
  Author: Özcan Acar <oezcanacar@kpmg.com>

      ilk commit: dosya1 eklendi
```

Maksadımız af599b1 hash değerindeki commitin 054671e hash değerindeki commiti baz alacak şekilde branch-a dalının yeniden yapılandırılması. Bunu gerçekleştirmek için rebase komutunu şu şekilde uygulayabiliriz.

```
// Branch-a dalına geçiş
[oacar@lin projem2]$ git checkout branch-a

// Rebase işlemi
[oacar@lin projem2]$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: branch-a birinci commit: metin1 eklendi
Applying: branch-a ikinci commit: metin2 eklendi
[oacar@lin projem2]$
```

Rebase işleminden sonra branch-a commit listesine bir göz atalım:

```
[oacar@lin projem2]$ git log --graph
* commit aed2ecf567f7a6c8abaf94948ad92981e0b040e2
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date: Thu Aug 13 11:18:23 2015 +0200
|
|     branch-a ikinci commit: metin2 eklendi
|
* commit 1ca11bc69ff11da435dc131fc77fc9b28c631270
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date: Thu Aug 13 11:17:50 2015 +0200
|
```

```
|      branch-a birinci commit: metin1 eklendi
|
* commit 054671e561325e125c643ace6d73413f0e63b5be
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date:   Thu Aug 13 11:19:52 2015 +0200
|
|      dördüncü commit: dosya4 eklendi
|
* commit babaf11f65a533288176ffabc3991b6733f22ec0
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date:   Thu Aug 13 11:19:05 2015 +0200
|
|      üçüncü commit: dosya3 eklendi
|
* commit 2d0ce05a41727754609e53e12b89d17c6f080b7f
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date:   Thu Aug 13 11:18:49 2015 +0200
|
|      ikinci commit: dosya2 eklendi
|
* commit af599b18088eda3c01e1856329daaaea2ecbdf0
| Author: Özcan Acar <oezcanacar@kpmg.com>
| Date:   Thu Aug 13 11:16:26 2015 +0200
|
|      ilk commit: dosya1 eklendi
```

Görüldüğü gibi rebase işleminden sonra branch-a bünyesindeki commitler 054671e commitini temel almaktadır. Bu master dalında yer alan en son committir. Bunun yanı sıra branch-a dalında yapmış olduğumuz commitlere yeni hash değerleri atandı, çünkü sahip oldukları baş (parent) değişti.

Git rebase işlemi yeni commit hash değerleri oluşturarak, deponun tarihçesini (history) değiştirmektedir. Ortak kullanılan depolarda rebase işlemi ile deponun tarihçesinin değiştirilmesi, referans niteliğinde olan eski commitlerin bulunmasını ve geçmişte yapılan değişikliklerin takip edilmesini güçleştirecektir. Bu sebepten dolayı rebase işlemi özel (private local) depolar ve dallar haricinde kullanılmamalıdır.

Git rebase işlemini interaktif olarak yapmak da mümkündür. Bu genelde çalışılan dal üzerindeki commit geçmişini yeniden düzenlemek için seçilen yöntemdir. Kod yazarken yapılan commitler yapılan işin parçalarını yansıtmayabilir. Dal üzerinde iş tamamlandıktan sonra, interaktif rebase ile dal geçmişi yeniden düzenlenebilir. Örneğin bu işlem esnasında gereksiz commitler yok edilebilir ve commit sırası düzenlenebilir. Bu işlemler ardından dal, ana proje dalı ile ve onun resmi geçmişi ile birleştirilmeye hazır hale getirilir.

İnteraktif rebase işlemini şu şekilde gerçekleştirebiliriz:

```
// Branch-a dalına geçiş
[oacar@lin projem2]$ git checkout branch-a

// İnteraktif rebase
[oacar@lin projem2]$ git rebase -i master
```

Bu komutun ardından Git bir editör içinde (vi) aşağıdaki içeriği sunacaktır:

```
pick 7b600fb branch-a birinci commit: metin1 eklendi
pick 8b7ce7d branch-a ikinci commit: metin2 eklendi

# Rebase 054671e..8b7ce7d onto 054671e
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Branch-a dalındayız ve bu dalı master dalı ile interaktif bir şekilde rebase yapmak istiyoruz. Editör içinde pick ibaresi ile başlayan iki satır görmekteyiz. Bunlar branch-a dalında yaptığımız commitlerdir. Eğer birinci commiti (7b600fb) yok etmek istersek, içeriği şu şekilde değiştirip, editörden çıkmamız gerekmektedir.

```
pick 8b7ce7d branch-a ikinci commit: metin2 eklendi
```

Editörden çıktıktan sonra Git rebase işlemini tamamlayacaktır. Şimdi dalın geçmişine bakarak, rebase işleminin nasıl tamamlandığında bir göz atalım:

```
[oacar@lin projem2]$ git log --oneline
104a62e branch-a ikinci commit: metin2 eklendi
```

```
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
af599b1 dosya1 eklendi
```

Görüldüğü gibi branch-a üzerinde yaptığımız ilk commit listede yer almamaktadır. Git burada sanki o commit hic yapılmamış gibi rebase işlemini gerçekleştirmiştir.

Eğer commit sırasını değiştirmek istiyorsak, editörün içeriğini şu şekilde değiştirebiliriz:

```
pick 8b7ce7d branch-a ikinci commit: metin2 eklendi
pick 7b600fb branch-a birinci commit: metin1 eklendi
```

Bu durumda rebase işlemi esnasında ikinci commit birinci committen önce yer alacaktır.

```
[oacar@lin projem2]$ git rebase -i master
Successfully rebased and updated refs/heads/branch-a.
[oacar@lin projem2]$ git log --oneline
a60e77f branch-a birinci commit: metin1 eklendi
04cf0ab branch-a ikinci commit: metin2 eklendi
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
af599b1 dosya1 eklendi
```

İki commiti birleştirip, yeni bir commit yapmak istediğimizi düşünelim. Bu durumda editörün içeriği şu şekilde olacaktır:

```
squash 7b600fb branch-a birinci commit: metin1 eklendi
squash 8b7ce7d branch-a ikinci commit: metin2 eklendi
```

Editörden çıktığımızda, söyle bir hata mesajı ile karşılaşmamız muhtemel:

```
[oacar@lin projem2]$ git rebase -i master
Cannot 'squash' without a previous commit
```

İki commiti birleştirebilmek için daha önce oluşturduğumuz en az bir commitin olması gerekmektedir. Squash ile işaretlediğimiz commitler mevcut olan bu commite dahil edileceklerdir. Bu yüzden yeni bir commit oluşturarak, rebase işlemini tekrar edelim.

```
// Dalın tarihçesi
[oacar@lin projem2]$ git log --oneline
8b7ce7d branch-a ikinci commit: metin2 eklendi
```

```

7b600fb branch-a birinci commit: metin1 eklendi
af599b1 dosya1 eklendi

// Yeni bir dosya
[oacar@lin projem2]$ touch metin3

// Dosyanın indexe eklenmesi
[oacar@lin projem2]$ git add --all

// Commit işlemi
[oacar@lin projem2]$ git commit -m "branch-a üçüncü commit: metin3 eklendi"
[branch-a f61ae86] branch-a üçüncü commit: metin3 eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 metin3

// Deponun tarihçesi
[oacar@lin projem2]$ git log --oneline
f4fb238 branch-a üçüncü commit: metin3 eklendi
8b7ce7d branch-a ikinci commit: metin2 eklendi
7b600fb branch-a birinci commit: metin1 eklendi
af599b1 dosya1 eklendi

// Rebase işlemi
[oacar@lin projem2]$ git rebase -i master
It seems that there is already a rebase-merge directory, and
I wonder if you are in the middle of another rebase.  If that is the
case, please try
    git rebase (--continue | --abort | --skip)
If that is not the case, please
    rm -fr "c:/Users/oezcanacar/projem2-bak/.git/rebase-merge"
and run me again.  I am stopping in case you still have something
valuable there.

```

Daha önce interaktif rebase oturumunu başlattığımız için yukarıda gördüğümüz hatayı aldık. Abort komutu ile bu oturumu sonlandırabiliriz, continue ile devam ettirebiliriz.

```

[oacar@lin projem2]$ git rebase --abort
[oacar@lin projem2]$ git rebase -i master

```

Başlattığımız yeni rebase oturumunda editörün içeriği şu şekilde olacaktır:

```

pick 7b600fb branch-a birinci commit: metin1 eklendi
pick 8b7ce7d branch-a ikinci commit: metin2 eklendi
pick af8f632 branch-a üçüncü commit: metin3 eklendi

```

Şimdi af8f632 ve 8b7ce7d commitlerini yok ederek, içeriklerini 7b600fb hash değerindeki ilk commite ekleyelim:

```
pick 7b600fb branch-a birinci commit: metin1 eklendi
squash 8b7ce7d branch-a ikinci commit: metin2 eklendi
squash af8f632 branch-a üçüncü commit: metin3 eklendi
```

Editörü sonlandırdığımız andan itibaren, Git rebase işlemini devam ettirir.

```
# This is a combination of 3 commits.
# The first commit's message is:
branch-a birinci commit: metin1 eklendi

# This is the 2nd commit message:

branch-a ikinci commit: metin2 eklendi

# This is the 3rd commit message:

branch-a üçüncü commit: metin3 eklendi
```

Yeni açılan editörde commit mesajını değiştirme fırsatı bulmaktayız. Yeni commit mesajını şu şekilde oluşturabiliriz:

```
# This is a combination of 3 commits.
# The first commit's message is:
rebase yaparak, son iki commiti yok ettim.
```

Editörü sonlandırdıktan sonra, rebase işlemi gerçekleşecek ve dalın yeni geçmişi şu şekilde olacaktır:

```
[oacar@lin projem2]$ git log --oneline
6a8e876 rebase yaparak, son iki commiti yok ettim.
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
af599b1 dosya1 eklendi
```

Şimdi mevcut bir commiti iki parçaya bölmek istediğimizi düşünelim. Çıkış noktamız şu şekilde:

```
[oacar@lin projem2]$ git log --oneline
8b7ce7d branch-a ikinci commit: metin2 eklendi
7b600fb branch-a birinci commit: metin1 eklendi
```

```
af599b1 dosya1 eklendi
```

Şimdi interaktif rebase seansını başlatalım:

```
[oacar@lin projem2]$ git rebase -i master
```

Açılan editörün içeriği şu şekilde olacaktır:

```
pick 7b600fb branch-a birinci commit: metin1 eklendi
pick 8b7ce7d branch-a ikinci commit: metin2 eklendi
```

8b7ce7d hash değerindeki ikinci commiti iki parçaya bölmek istediğimiz için editörün içeriğini şu şekilde değiştiriyoruz:

```
pick 7b600fb branch-a birinci commit: metin1 eklendi
edit 8b7ce7d branch-a ikinci commit: metin2 eklendi
```

Editörü kapattıktan sonra karşılaçağımız içerik şu şekilde olacaktır:

```
[oacar@lin projem2]$ git rebase -i master
Stopped at 8b7ce7d1dfa9d9ef95fd753a15fcd43516d61a29...
        branch-a ikinci commit: metin2 eklendi
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue
```

Şimdi bu commit bünyesinde değişikliğe uğrayan dosyaları görebilmek için aşağıdaki şekilde reset işlemini gerçekleştirmemiz gerekiyor. Bu bizi 8b7ce7d öncesinde üzerinde çalıştığımız dosyaların son haline getirecektir:

```
[oacar@lin projem2]$ git reset HEAD^
```

Status komutu ile hangi dosyaların indexe eklenmek için beklediklerini görebiliriz:

```
[oacar@lin projem2]$ git status
rebase in progress; onto 054671e
You are currently editing a commit while rebasing branch 'branch-a'
        on '054671e'.

(use "git commit --amend" to amend the current commit)
```



```
(use "git rebase --continue" önce you are satisfied with your changes)

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

        metin2

nothing added to commit but untracked files present (use "git add" to track)
```

Metin2 ismini taşıyan dosya commit işlemi için bekliyor. Yeni bir dosya daha açarak, bu iki dosya için yeni bir commit oluşturalım:

```
[oacar@lin projem2]$ touch metin3
[oacar@lin projem2]$ git add --all
[oacar@lin projem2]$ git commit -m "metin2 ve metin3 dosyaları eklendi"
```

Tekrar yeni bir dosya oluşturarak, bu dosyayı commitliyoruz:

```
[oacar@lin projem2]$ touch metin4
[oacar@lin projem2]$ git add --all
[oacar@lin projem2]$ git commit -m "metin4 dosyası eklendi"
```

Rebase işlemine --continue parametresiyle devam ediyoruz:

```
[oacar@lin projem2]$ git rebase --continue
```

Şimdi oluşan sonuca bir göz atalım:

```
[oacar@lin projem2]$ git log --oneline
08fe958 metin4 dosyası oluşturuldu
353c219 metin2 ve metin3 dosyaları oluşturuldu
5ea81a6 branch-a birinci commit: metin1 eklendi
054671e dördüncü commit: dosya4 eklendi
babaf11 üçüncü commit: dosya3 eklendi
2d0ce05 ikinci commit: dosya2 eklendi
af599b1 dosya1 eklendi
```

Görüldüğü gibi branch-a dalındaki ikinci commiti iki parçaya bölerek, 353c219 ve 08fe958 hash değerlerinde iki yeni commit oluşturduk.

Git Reset

İkinci bölümde git revert komutu ile tanışmıştık. Revert komutu ile yaptığımız değişiklikleri geri alabilmekteyiz. Revert komutu kullanıldığında mevcut bir commit ile yapılan değişiklikler geriye alınabilmekte, lakin böyle bir işlemin gerçekleştiği yeni bir commit ile deponun tarihçesine yazılmaktadır. Bu yüzden revert işlemini güvenli bir işlem olarak düşünebiliriz, çünkü hangi işlemin gerçekleştiğini deponun tarihçesine bakarak, görmek mümkündür. Reset komutunda durum biraz farklı.

Reset komutu deponun tarihçesinde bulunan commit nesnelerini yok etmek için kullanılmaktadır. Reset işlemini gerçekleştirmek için belli bir commit nesnesini seçmemiz gerekiyor. Bu commit nesnesi üzerinde reset işlemini gerçekleştirdiğimiz andan itibaren, sanki bir kurdeleyi kesmişiz gibi, o commit nesnesinden sonra gelen commitlere olan bağımızı kaybederiz. Reset işleminden sonra deponun belli bir tarihçesini kaybetme riski bulunmaktadır. Reset işleminin nasıl yapıldığını şimdi bir örnek üzerinde inceleyelim.

Kullandığımız deponun commit listesi şu şekildedir:

```
[oacar@lin projem]$ git log --oneline
b4073a6 dördüncü commit
3fa8f0d üçüncü commit
d113b24 ikinci commit
fe02106 ilk commit
[oacar@lin projem]$
```

b4073a6 hash değerine sahip commiti listeden yani deponun tarihçesinden silmek istediğimizi düşünelim. Bunu gerçekleştirmek için reset komutunu şu şekilde kullanabiliriz:

```
[oacar@lin projem]$ git reset 3fa8f0d
[oacar@lin projem]$ git log --oneline
3fa8f0d üçüncü commit
d113b24 ikinci commit
fe02106 ilk commit
[oacar@lin projem]$
```

Dördüncü commiti listeden silebilmek için bir önceki commitin hash değerini reset komutuna parametre olarak vermemiz gerekiyor. Reset işlemi gerçekleştikten sonra commit listesinde dördüncü commitin artık yer almadığını görmekteyiz.

Mevcut bir commiti sildikten sonra, bu commitin ihtiva ettiği değişiklikleri bir şekilde yönetmemiz gerekmektedir. Ya bu değişiklikleri tamamen geriye alabiliriz ya da başka bir commit bünyesinde depoya ekleyebiliriz. Yok ettiğimiz commit ile depo dışında kalan dosyalara şu şekilde ulaşabiliriz:

```
[oacar@lin projem]$ git status
HEAD detached from e25a447
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        dosya4

nothing added to commit but untracked files present (use "git add" to track)
[oacar@lin projem]$
```

Görüldüğü gibi reset ile yok ettiğimiz commit bünyesinde dosya4 ismini taşıyan bir dosya yer almaktaymış. Bu dosyanın ve reset sonunda ortada kalan değişikliklerin akibeti ile kullanıcı olarak bizim ilgilenmemiz gerekmektedir.

Şimdi bu dosyayı add ile depo index ya da staging area olarak tanımladığımız alana alalım. Commit öncesinde depoya eklemek istediğimiz dosyalar bu alanda olmak zorunda.

```
// Dosya4 indexe eklenir
[oacar@lin projem]$ git add dosya4

// Çalışma alanının son durumu
[oacar@lin projem]$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   dosya4

[oacar@lin projem]$ cat .git/refs/heads/master
```

Add komutu ile dosya4 şimdi indexe eklenmiş oldu. Reset komutunu aynı zamanda indexi boşaltmak ya da temizlemek için de kullanabiliriz. Bu yaptığımız değişiklikleri yeniden yapılandırmak için gerekli olabilir. Örneğin indexe eklediğimiz dosya4 isimli dosyası buradan çıkarmak için reset komutunu şu şekilde kullanabiliriz:

```
// Reset işlemi indexi boşaltır
[oacar@lin projem]$ git reset

// Çalışma alanının son durumu
[oacar@lin projem]$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        dosya4
```

```
nothing added to commit but untracked files present (use "git add" to track)
[oacar@lin projem]$
```

Görüldüğü gibi dosya4 isimli dosyayı indexden çıkartmış olduk.

Reset komutu parametre olmadan kullanıldığında, index içindeki depoya eklenmeyi bekleyen tüm dosyalar buradan çıkartılır. Örneğin indexe eklenmeyi bekleyen dosya4 ve dosya5 isimlerinde iki dosyamız olsun:

```
[oacar@lin projem]$ git status
HEAD detached from e25a447
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        dosya4
        dosya5

nothing added to commit but untracked files present (use "git add" to track)
```

Bu dosyaları şimdi indexe ekliyoruz:

```
[oacar@lin projem]$ git add --all
[oacar@lin projem]$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   dosya4
        new file:   dosya5

[oacar@lin projem]$
```

Git status yeni dosyaların indexe eklendiğini gösteriyor. Reset komutu ile bu dosyaları indexten çıkarabiliriz:

```
[oacar@lin projem]$ git reset
[oacar@lin projem]$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        dosya4
        dosya5

nothing added to commit but untracked files present (use "git add" to track)
[oacar@lin projem]$
```

Görüldüğü gibi her iki dosyayı reset komutu ile tekrar indexden silmeyi başardık. Şimdi bu iki dosyayı tekrar indexe ekleyelim:

```
[oacar@lin projem]$ git add --all
[oacar@lin projem]$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   dosya4
        new file:   dosya5
```

Yaptığımız değişiklikleri tamamen ortadan kaldırmak ve içinde çalıştığımız dizine en son commitin ihtiva ettiği dosyaları almak için --hard parametresini şöyle kullanabiliriz:

```
[oacar@lin projem]$ git reset --hard
HEAD is now at 3fa8f0d üçüncü commit

[oacar@lin projem]$ git status
nothing to commit, working directory clean
[oacar@lin projem]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 24 16:25 dosya1
-rw-r----- 1 oacar users 0 Apr 24 16:32 dosya2
-rw-r----- 1 oacar users 0 Apr 24 16:33 dosya3
[oacar@lin projem]$
```

Aynı şekilde belli bir commite şu şekilde geri dönebiliriz (ondan sonraki commit nesneleri ile olan bağ kopar):

```
[oacar@lin projem]$ git reset --hard <commit-hash>
```

Bu durumda index boşaltılacak ve tüm çalışma alanı <commit-hash> değerindeki commitin sahip olduğu içerik seviyesine getirilecektir. Bu aynı zamanda çalışma alanındaki (working copy) commitlenmemiş olan dosyaların silinmesi anlamına gelmektedir, yani reset öncesi yaptığımız tüm değişiklikleri reset operasyonu ile kaybetmiş olmaktadır.

Verdiğim örneklerde görüldüğü gibi reset komutu ile hem index üzerinde hem de deponun tarihçesi üzerinde değişiklik yapılabilir. Reset işlemi veri kaybına sebep olabilir.

Git Clean

Clean komutu ile dizin içinde bulunan ve indexe henüz eklenmemiş tüm değişiklikleri yok edebiliriz. Bu temizlik işleminin nasıl yapıldığını bir örnek üzerinde inceleyelim.

Önce iki yeni dosya oluşturalım:

```
[oacar@lin projem]$ touch dosya4

[oacar@lin projem]$ touch dosya5

[oacar@lin projem]$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        dosya4
        dosya5

nothing added to commit but untracked files present (use "git add" to track)
```

Bu dosyaları doğrudan silmeden önce, hangi dosyaların silineceğini şu şekilde öğrenebiliriz:

```
[oacar@lin projem]$ git clean -n
Would remove dosya4
Would remove dosya5
[oacar@lin projem]$
```

Şu şekilde bu iki dosyayı silebiliriz:

```
[oacar@lin projem]$ git clean -f
Removing dosya4
Removing dosya5

[oacar@lin projem]$ git status
nothing to commit, working directory clean
[oacar@lin projem]$
```

Dizin ve içinde bulunan dosyaları şu şekilde silebiliriz:

```
[oacar@lin projem]$ mkdir dizin1

[oacar@lin projem]$ cd dizin1/

[oacar@lin dizin1]$ touch dosya1

[oacar@lin dizin1]$ cd ..
```

```
[oacar@lin projem]$ git clean -df
Removing dizin1/
[oacar@lin projem]$
```

Bir dizini ve içindekileri şöyle de silebiliriz:

```
[oacar@lin projem]$ mkdir dizin1

[oacar@lin projem]$ git clean -f dizin1/

[oacar@lin projem]$ git status
nothing to commit, working directory clean
```

Ignore listesinde olan tüm dosya ve dizinleri şu şekilde silebiliriz:

```
[oacar@lin projem]$ git clean -xf

[oacar@lin projem]$ git status
nothing to commit, working directory clean
[oacar@lin projem]$
```

Git Checkout

Checkout komutu ile daha önce tanışmıştık. Dallar arası geçiş yapmak için checkout komutunu kullandık. Checkout komutunu dal geçişleri yanı sıra commitler arası ve dosyanın değişik versiyonları arasında geçiş için de kullanabiliriz. Şimdi bunun nasıl yapıldığını bir örnek üzerinde inceleyelim.

Çıkış noktamız şu şekilde olsun:

```
[oacar@lin projem]$ ls -l
total 4
-rw-r----- 1 oacar users 5 Apr 27 13:27 dosya1
-rw-r----- 1 oacar users 0 Apr 27 13:27 dosya2
[oacar@lin projem]$
```

Çalıştığımız dizinde iki dosya yer alıyor. Bu dosyalar üzerinde yapılan işlemlere göz atalım:

```
[oacar@lin projem]$ git log --oneline
0dd227f dosya1 için ikinci commit
16c3356 dosya2 için ilk commit
```

```
40dea6c dosya1 için ilk commit
[oacar@lin projem]$
```

Dosya1 isimli dosya üzerinde iki işlem gerçekleştirilmiş. Bu dosyanın güncel içeriği şu şekilde:

```
[oacar@lin projem]$ cat dosya1
test
[oacar@lin projem]$
```

Odd227f hash değerine sahip commit bünyesinde dosya1 üzerinde işlem yapıldığını görmekteyiz. Nasıl bir işlem yapıldığını görmek için commitin içeriğine bakıyoruz:

```
[oacar@lin projem]$ git show Odd227f
commit Odd227f42f53af0325c191d47948c164901fc0d3
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 27 13:27:37

    dosya1 için ikinci commit

diff --git a/dosya1 b/dosya1
index e69de29..9daeafb 100644
--- a/dosya1
+++ b/dosya1
@@ -0,0 +1 @@
+test
[oacar@lin projem]$
```

Bu dosyanın ilk committeki haline edinmek istersek, checkout komutunu şu şekilde kullanabiliriz:

```
[oacar@lin projem]$ git checkout 40dea6c dosya1

[oacar@lin projem]$ cat dosya1
[oacar@lin projem]$

[oacar@lin projem]$ git show 40dea6c
commit 40dea6c51af43d81e5593cfdbfd631e805ac8a70
Author: Özcan <acar@agilementor.com>
Date:   Mon Apr 27 13:26:57

    dosya1 için ilk commit

diff --git a/dosya1 b/dosya1
new file mode 100644
index 0000000..e69de29
[oacar@lin projem]$
```


Görüldüğü gibi ilk commitde dosya1 isimli dosya sadece oluşturulmuş (new file mode 100644), lakin bir içeriğe sahip değildir. Bu dosyanın en son haline geri dönmek için checkout komutunu şu şekilde kullanabiliriz:

```
[oacar@lin projem]$ git checkout Odd227f dosya1
[oacar@lin projem]$ cat dosya1
test
[oacar@lin projem]$
```

Checkout komutuyla bir dosyanın tanımladığımız bir commit bünyesindeki haline erisebilmekteyiz. Bu dosya checkout işlemi esnasında depodan alınmakta ve içinde çalıştığımız dizindeki diğer versiyonuyla değiştirilmektedir. Checkout komutunun bu tarz kullanımında içinde çalıştığımız dizin değişikliğe uğramaktadır. Checkout işlemi esnasında edindiğimiz dosyalar üzerinde değişiklik yapıp, onları indexe ekledikten sonra, yeni commitler bünyesinde depoya ekleyebiliriz.

Checkout komutunu aynı zamanda tüm bir commitin içeriğini edinmek için de kullanabiliriz. Bu durumda tanımladığımız commite ait tüm dosyalar depodan doğrudan çalıştığımız dizine aktarılırlar. Bunun nasıl uygulandığını görmek için içinde çalıştığımız dizine tekrar bir göz atalım:

```
[oacar@lin projem]$ ls -l
total 4
-rw-r----- 1 oacar users 5 Apr 27 14:04 dosya1
-rw-r----- 1 oacar users 0 Apr 27 14:04 dosya2

[oacar@lin projem]$ git log --oneline
Odd227f dosya1 için ikinci commit
16c3356 dosya2 için ilk commit
40dea6c dosya1 için ilk commit
[oacar@lin projem]$
```

İlk commite geri dönmek için şu işlemi gerçekleştiriyoruz:

```
[oacar@lin projem]$ git checkout 40dea6c
Note: checking out '40dea6c'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

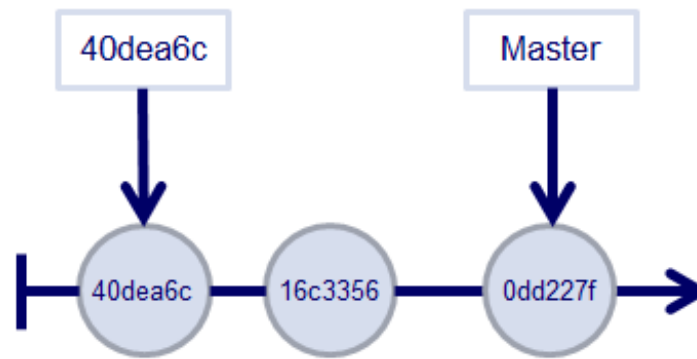
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b new_branch_name

HEAD is now at 40dea6c... dosya1 için ilk commit

[oacar@lin projem]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 27 14:09 dosya1
[oacar@lin projem]$
```

Çalıştığımız dizine baktığımızda, sadece dosya1 isimli dosyanın var olduğunu görmekteyiz. Bu deponun birinci commit sonrasındaki halidir ve bu durum çalıştığımız dizine yansımıştır. Şimdi istediğimiz şekilde dosya1 üzerinde değişikli yapabiliriz. Yapacağımız değişiklikler commiti ya da deponun içeriğini değiştirmez, çünkü kullandığımız depo detached modundadır. Detached mod ile Git dünyasında ana geliştirme dalında geri gidildiği ve bu durumda yapılacak yeni bir commitin nereye konulacağını bilinememesi ifade edilmektedir. Bu durumu resim 14 de görmekteyiz. Altıncı bölümde teknik olarak detached modun ne anlama geldiğini yakından inceleyeceğiz.



Resim 14

40dea6c hash değerindeki commite yatay geçiş yaptık. Bu durumda, HEAD olarak isimlendirilen ve deponun hangi dalı ve commiti üzerinde olduğumuzu gösteren gösterçek bu commite işaret etmektedir. Bu göstergecin içeriğine şu şekilde ulaşabiliriz:

```
[oacar@lin projem]$ cat .git/HEAD
40dea6c51af43d81e5593cfdbfd631e805ac8a70
[oacar@lin projem]$
```

HEAD görüldüğü gibi 40dea6c hash değerindeki commite işaret etmektedir. Şimdi ana geliştirme dalı olan master dalının hangi commite işaret ettiğine bir göz atalım:

```
[oacar@lin projem]$ cat .git/refs/heads/master
0dd227f42f53af0325c191d47948c164901fc0d3
[oacar@lin projem]$
```

Görüldüğü gibi master 0dd227f hash değerindeki commite işaret etmektedir. Checkout ile başka bir commite yatay geçiş yaptıktan yani detached moda düştükten sonra depodan edindiğimiz dosyalar üzerinde işlem yapmamız anlamsız hale gelmektedir, çünkü yeni bir commitin oluşturulması mümkün değildir. Yeni bir commit sadece üzerinde çalışılan dalın en son commitine mütakiben yapılabilir (aslında burada teknik olarak bir commit oluşturmak mümkün, lakin bu commit havala asılı kalır, çünkü hiçbir dala ait değildir).

Ana dalın son commitine şu şekilde yatay geçiş yapabiliriz:

```
[oacar@lin projem]$ git checkout master
Previous HEAD position was 40dea6c... dosyal için ilk commit
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[oacar@lin projem]$
```

Şimdi HEAD göstergesinin değerine bir göz atalım:

```
[oacar@lin projem]$ cat .git/HEAD
ref: refs/heads/master
[oacar@lin projem]$
```

Görüldüğü gibi şimdi HEAD refs/heads/master dosyasındaki değere işaret etmektedir. Bu değer master dalının hangi commite işaret ettiğinin bilgisidir ve şu şekildedir:

```
[oacar@lin projem]$ cat .git/refs/heads/master
0dd227f42f53af0325c191d47948c164901fc0d3
[oacar@lin projem]$
```

Bu son değişiklik ile master tekrar son commite işaret etmektedir.

Eğer yatay geçiş yaptığımız diğer commit bünyesindeki dosyalar üzerinde değişiklik yaparsak, tekrar master dalına kolay kolay dönemeyiz. Bunun nedenini tekrar bir örnek üzerinde inceleyelim. Örneğin tekrar 40dea6c hash değerindeki commite geçtiğimizi düşünelim:

```
[oacar@lin projem]$ git checkout 40dea6c
Note: checking out '40dea6c'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:
```

```
git checkout -b new_branch_name
```

```
HEAD is now at 40dea6c... dosya1 için ilk commit
[oacar@lin projem]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 27 15:23 dosya1
[oacar@lin projem]$
```

Detached moda düştük. İçinde çalıştığımız dizinde (working copy) dosya1 isimli dosya var. Bu dosyanın içeriğini şimdi değiştiriyoruz:

```
[oacar@lin projem]$ echo "test" >dosya1
```

Bu andan itibaren git checkout master ile ana dala geçmemiz mümkün değildir, çünkü mevcut bir dosyayı değiştirdik:

```
[oacar@lin projem]$ git checkout master
error: Your local changes to the following files would be overwritten
        by checkout:
        dosya1
Please, commit your changes or stash them before you can switch branches.
Aborting
[oacar@lin projem]$
```

Burada teknik olarak yapılabilecek üç işlem vardır. Bunlar:

- Daha sonra inceleyeceğimiz stash komutu ile dosya1 üzerinde yaptığımız değişiklikleri rafa kaldırıp, master dalına geçiş için oluşmuş engeli kaldırabiliriz, ya da
- Yeni bir commit oluşturup, yaptığımız değişikliği deponun veri tabanına ekleyebiliriz. Detached modda olduğumuz için bu commit işimize yaramayacaktır. Bu commiti çıkmaz bir sokak gibi düşünebiliriz. Dal üzerindeki en son commit nesnesine geçtiğimiz andan itibaren, oluşturduğumuz bu commit ile olan bağımızı kaybederiz, çünkü oluşturduğumuz commit nesnesinin hiçbir dal ile bağı yoktur.
- Reset komutu ile yaptığımız değişiklikleri aşağıdaki şekilde geri alabiliriz.

```
[oacar@lin projem]$ git reset --hard HEAD is now at 40dea6c dosya1 için ilk commit
[oacar@lin projem]$
```

Şimdi ana dala geçiş yapabiliriz:

```
[oacar@lin projem]$ git checkout master
Previous HEAD position was 40dea6c... dosyal için ilk commit
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[oacar@lin projem]$
```

Bunun yanı sıra detached modda olsak bile yaptığımız değişikliği commitleyebiliriz:

```
[oacar@lin projem]$ git checkout 40dea6c
[oacar@lin projem]$ echo "test" >> dosyal
[oacar@lin projem]$ git add dosyal
[oacar@lin projem]$ git commit -m "test"
[detached HEAD 86d3672] test
1 file changed, 1 insertion(+)
[oacar@lin projem]$ git log --oneline
86d3672 test
40dea6c dosyal için ilk commit
[oacar@lin projem]$
```

Ama daha öncede bahsettiğim gibi bu commit anlamsız olurdu, çünkü o ana geliştirme dalı ya da başka bir dala bağlı değil.

Şimdi tekrar ana dala geçmeye çalışalım:

```
[oacar@lin projem]$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

    86d3672 test

If you want to keep them by creating a new branch, this may be a good time
to do so with:

    git branch new_branch_name 86d3672

Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[oacar@lin projem]$
```

Git'in verdiği mesaj "you are leaving 1 commit behind, not connected to any of your branches" yeterince açıklayıcı nitelikte sanırım. Git bize detached moddayken oluşturulan commitin hic bir yere bağlı olmadığını, ama yeni bir dala aktarılarak, kaybolmaktan korunabileceğini söylemektedir. Buradan şu sonuçları çıkarabiliriz:

- Detached moddayken dosyalar üzerinde deęişiklik yapmak anlamlı deęildir.
- Detached modda dosyaları sadece salt-okunur (read-only) olarak idrak etmeliyiz.
- Detached modda yaptığımız deęişiklikleri reset komutu ile geri alabiliriz.

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Branch (dal ya da kol) konseptini kullanarak, birbirinden bağımsız uygulama özelliklerini paralel olarak implemente edebiliriz.
- Git checkout komutu ile hem yeni bir dal oluşturmak hem de mevcut bir dala yan geçiş yapmak mümkündür.
- Git branch komutu ile depoda yer alan dalların listesi edinilebilir.
- Git merge komutu ile dalların içerikleri birleştirilebilir.
- Git reflog komutu ile yerel bir deponun tarihçesini yani dal bünyesinde olup bitenlerin listesini edinmek mümkündür.
- Rabase işlemi esnasında fast-forward-merge yöntemiyle mevcut dalın temel aldığı daldaki değişiklikler mevcut dala aktarılır. Mevcut dal üzerinde yapılan tüm değişiklikler bu merge işlemi esnasında oluşan en son commit nesnesini baz alacak şekilde yeniden yapılandırılırlar. Rabase işleminin başlıca sebebi, bir uygulama özelliğini implemente ederken oluşan değişiklikleri ana dalda meydana gelen değişiklikler üzerine inşa etme arzusudur.
- Git revert komutu ile deponun tarihçesini değiştirmek mümkün iken, git reset komutu ile mevcut çalışma alanının (working copy) ve indexin içeriğini değiştirmek mümkündür.
- Clean komutu ile dizin içinde bulunan ve indexe henüz eklenmemiş tüm değişiklikleri yok edebiliriz.

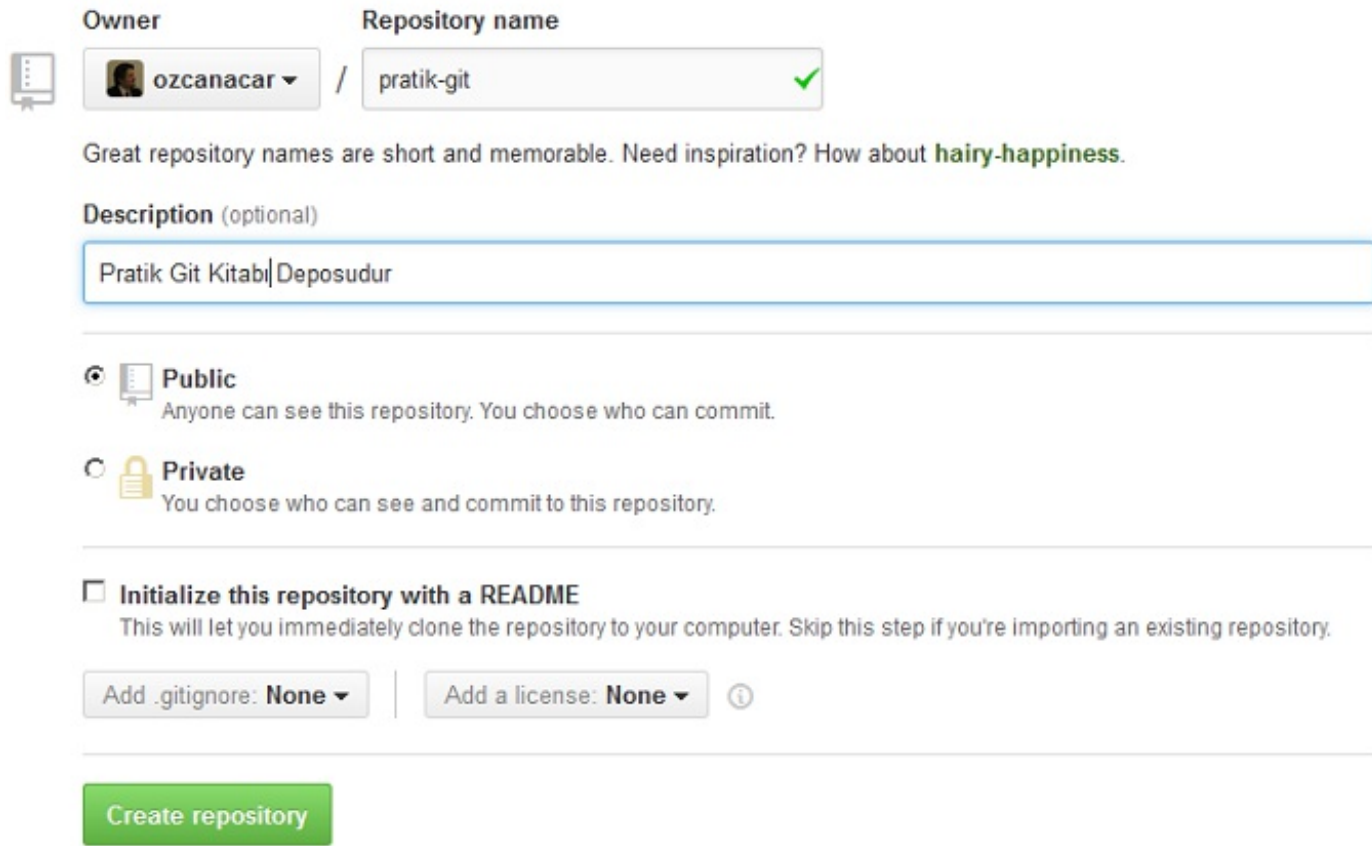
4. Bölüm

Remote Depo Kullanımı

Git Remote

Git remote komutu ile mevcut iki depoyu veri alışverişi yapacak şekilde ilişkilendirebiliriz. Bunun nasıl yapıldığını GitHub.com servisini kullanarak, göstermek istiyorum. GitHub birçok açık kaynaklı uygulamanın kullandığı Git depo oluşturma ve kullanma hizmeti sunan bir servistir. Git bash ve GitHub aracılığı ile bu bölümde yer alan örnekleri uygulayabilirsiniz.

GitHub için gerekli hesabı oluşturduktan sonra, <https://github.com/new> adresine giderek, yeni bir depo oluşturabiliriz:



Owner: ozcanacar / Repository name: pratik-git ✓

Great repository names are short and memorable. Need inspiration? How about **hairy-happiness**.

Description (optional): Pratik Git Kitabı Deposudur

☒ Public: Anyone can see this repository. You choose who can commit.

☐ Private: You choose who can see and commit to this repository.

☒ Initialize this repository with a README: This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Resim 1

Pratik-git ismini taşıyan boş bir depo oluşturdum.

Quick setup — if you've done this kind of thing before

Set up in Desktop

or

HTTPS

SSH

<https://github.com/ozcanacar/pratik-git.git>

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo # pratik-git >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/ozcanacar/pratik-git.git
git push -u origin master
```

**...or push an existing repository from the command line**

```
git remote add origin https://github.com/ozcanacar/pratik-git.git
git push -u origin master
```

**...or import code from another repository**

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Resim 2

GitHub.com üzerinde oluşturduğumuz depoyu klonlamadan, bu deponun içeriğini mevcut yerel bir depo ile ilişkilendirebiliriz. Bu ilişkiyi oluşturmak için önce yerel bir depo oluşturmamız gerekiyor. Yerel depoyu oluşturmadan önce, deponun içinde yer alacağı dizini şu şekilde tanımlıyoruz:

```
[oacar@lin repo]$ mkdir pratik-git
[oacar@lin repo]$ ls -l
total 4
drwxr-x--- 2 oacar users 4096 Apr 27 15:55 pratik-git
[oacar@lin repo]$ cd pratik-git/
```

Bu dizin içinde yeni bir dosya oluşturuyorum:

```
[oacar@lin pratik-git]$ echo "#Pratik Git" >> README.md
```

Yerel depoyu oluşturun:

```
[oacar@lin pratik-git]$ git init
Reinitialized existing Git repository in /home/oacar/repo/pratik-git/.git/
```

Oluşturduğum dosyayı indexe (staging area) ekliyorum:

```
[oacar@lin pratik-git]$ git add README.md
```

Commit işlemini gerçekleştiriyorum:

```
[oacar@lin pratik-git]$ git commit -m "first commit"
[master (root-commit) b8dc59d] first commit
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

Bu commit yerel depomda gerçekleşti. Commitin Github'daki depo ile henüz bir ilişkisi yok. Şimdi iki depo arasındaki ilişkiyi oluşturun:

```
[oacar@lin pratik-git]$ git remote add origin https://github.com/ozcanacar/pratik-git.git
```

Yerel depo ile GitHub deposu arasındaki bağlantının ismi origin. Burada herhangi bir isim seçilebilir.

Şimdi yerel depomda yaptığım değişiklikleri GitHub sunucularında yer alan pratik-git isimli depoya aktarıyorum:

```
[oacar@lin pratik-git]$ git push -u origin master
Username for 'https://github.com': ozcanacar
Password for 'https://ozcanacar@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 227 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ozcanacar/pratik-git.git
* [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Git push işlemini gerçekleştirirken origin master bilgisini verdim, çünkü GitHub üzerinde oluşturduğum yeni deponun içinde herhangi bir dal bulunmuyor. Bu dalı oluşturmak ve yerel

depomuzda referans olarak kullanmak için origin master parametrelerini kullanıyoruz. Daha sonra gerçekleştireceğimiz bir push işleminde git push komutunu girmemiz yeterli olacaktır.

Git remote komutu yerel bir depo ile bir sunucudaki bir depo arasında veri alışverişini sağlamak için gerekli bağlantıyı oluşturmada kullanılmaktadır. Birden fazla depoya bu şekilde bağlantı kurmak mümkündür. Bu bağlantıları görmek için remote komutunu -v parametresi ile şu şekilde çalıştırabiliriz:

```
[oacar@lin pratik-git]$ git remote -v
origin  http://github.com/ozcanacar/pratik-git.git (fetch)
origin  http://github.com/ozcanacar/pratik-git.git (push)
[oacar@lin pratik-git]$
```

Tek bir sunucu kullanmamıza rağmen, iki origin kaydı görmekteyiz. İlk kayıt ile sunucu üzerindeki depodan değişikliklerin hangi adres üzerinden alınacağı (fetch), diğer kayıt ile sunucudaki depoya değişikliklerin hangi adres üzerinden gönderileceği tanımlanmaktadır. Örneğin fetch işlemi için http, pull işlemi için ssh protokolünü kullanabiliriz. Bu komutların kullanımını bu bölümde detaylı olarak inceleyeceğiz. Git clone komutunu kullanarak mevcut bir depoyu klonladığımızda, klonlanan depo hakkındaki bilgiler remote listesine otomatik olarak eklenmektedir. Bu şekilde klonlanan depo ile doğrudan push ve pull işlemleri gerçekleştirilebilmektedir.

Şu şekilde yeni bir bağlantı oluşturabiliriz:

```
[oacar@lin pratik-git]$ git remote add github http://github.com/ozcanacar/test.git
```

Tekrar remote bağlantı listesine baktığımızda:

```
[oacar@lin pratik-git]$ git remote -v
github  http://github.com/ozcanacar/test.git (fetch)
github  http://github.com/ozcanacar/test.git (push)
origin  http://github.com/ozcanacar/pratik-git.git (fetch)
origin  http://github.com/ozcanacar/pratik-git.git (push)
[oacar@lin pratik-git]$
```

Oluşturduğumuz ilk bağlantı origin, ikinci bağlantı github ismini taşımaktadır. Bu isimlere şu şekilde de ulaşabiliriz:

```
[oacar@lin pratik-git]$ git remote
github
origin
[oacar@lin pratik-git]$
```

Bir bağlantıyı şu şekilde silebiliriz:

```
[oacar@lin pratik-git]$ git remote rm origin
```

Bir bağlantının ismini şu şekilde değiştirebiliriz:

```
[oacar@lin pratik-git]$ git remote rename github github-test
[oacar@lin pratik-git]$ git remote
github-test
[oacar@lin pratik-git]$
```

Merkezi Depoya Yeni Bir Dal Nasıl Eklenir?

GitHub.com üzerinde yer alan pratik-git isimli depoya yeni bir dal eklemek istediğimizi düşünelim. GitHub'ın sunduğu arayüz üzerinden depoya yeni bir dal eklemek mümkün. Bunun yanı sıra bu işlemi git branch komutunu kullanarak, yerel depo aracılığı ile şu şekilde gerçekleştirebiliriz:

Öncelikle yerel depomuzda hangi dalların olduğuna bir göz atalım:

```
[oacar@lin pratik-git]$ git branch
* master
[oacar@lin pratik-git]$
```

Yerel depomuzda sadece master isminde bir dal bulunuyor. Remote komutu ile ilişki kurduğumuz GitHub deposundaki dalların listesini şu şekilde edinebiliriz:

```
[oacar@lin pratik-git]$ git branch -r
origin/master
[oacar@lin pratik-git]$ git branch -a
* master
remotes/origin/master
[oacar@lin pratik-git]$
```

Sunucu üzerindeki depoda da master isminde bir dal bulunuyor. Şimdi yerel depomuzda yeni bir dal oluşturalım:

```
[oacar@lin pratik-git]$ git checkout -b branch-a
Switched to a new branch 'branch-a'
```

Branch-a ismini taşıyan bu dala yeni bir dosya ekleyelim:

```
[oacar@lin pratik-git]$ touch test1  
[oacar@lin pratik-git]$ git add test1
```

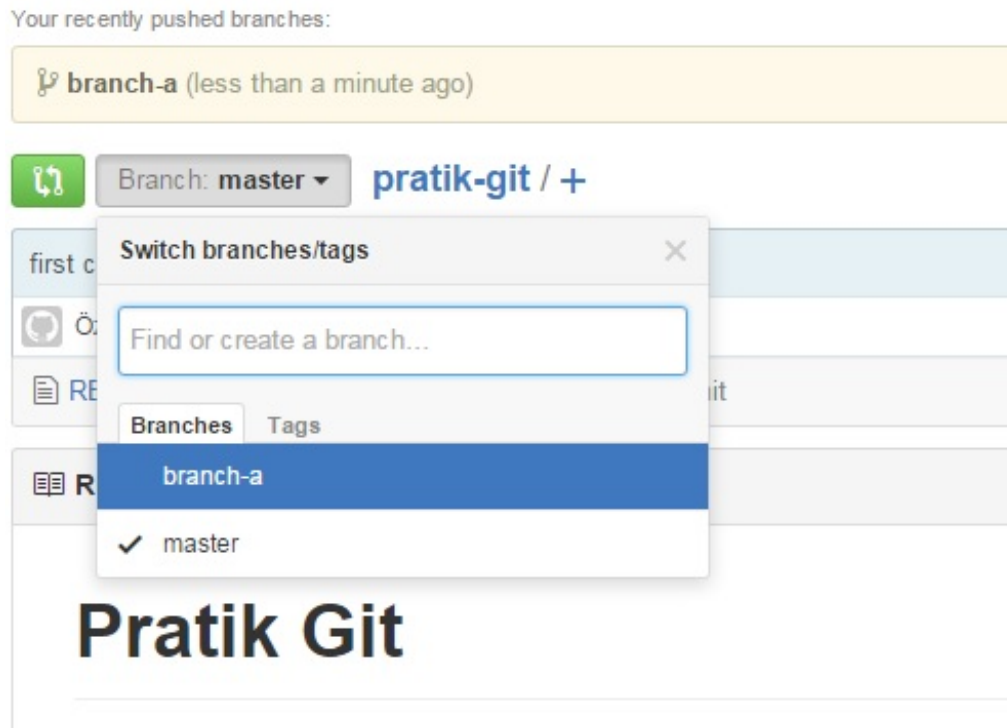
Ve bu değişikliği commitleyelim:

```
[oacar@lin pratik-git]$ git commit -m "test1 eklendi"  
[branch-a 2bf7d08] test1 eklendi  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 test1
```

Bu dalı sunucuya aktarmak için git push komutunu şu şekilde kullanabiliriz:

```
[oacar@lin pratik-git]$ git push origin branch-a  
Counting objects: 4, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 274 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To git@github.com:ozcanacar/pratik-git.git  
* [new branch]      branch-a -> branch-a
```

GitHub.com un kullanıcı arayüzünden baktığımızda, yeni dalın oluşturulduğunu görmekteyiz:



Resim 3

Bilindiği üzere origin GitHub.com da yer alan deponun adresine işaret etmektedir. Bu ismi daha önce git remote komutu ile tanımladık (clone komutu kullanıldıysa, bu kayıt otomatik olarak oluşturulur). Remote listemiz şu şekildedir:

```
[oacar@lin pratik-git]$ git remote -v
origin  http://github.com/ozcanacar/pratik-git.git (fetch)
origin  http://github.com/ozcanacar/pratik-git.git (push)
```

Yeni oluşturduğumuz branch-a ismini taşıyan dalı tüm içeriğiyle <http://github.com/ozcanacar/pratik-git.git> adresindeki depoya aktardık. Bunun için yerel deponun bir pratik-git.git klonu olması gerekmiyor. Git remote komutu ile herhangi bir Git deposuna bağlantı kurarak, yerel depoda mevcut bir dalı diğer depoya aktarabiliriz ya da diğer dosyadaki tüm dalları kendi yerel depomuza çekebiliriz.

Şimdi branch listemize tekrar bir göz atalım:

```
[oacar@lin pratik-git]$ git branch -a
* branch-a
  master
  remotes/origin/branch-a
  remotes/origin/master
```

Görüldüğü gibi yerel depomuzda branch-a ve master isminde iki dal, origin isimli depoda push komutuyla oraya gönderdiğimiz branch-a ve master dalları bulunmaktadır.

Git Fetch

Fetch komutu ile merkezi bir deponun sahip olduğu dalları kendi yerel depomuza çekebiliriz. Bu genelde merkezi depolar üzerinde diğer kullanıcılar tarafından yapılan değişiklikleri görmek için yapılan bir işlemdir. Dallar yerel depomuza eklenmekle birlikte, içinde çalıştığımız dizinde bir değişiklik oluşmamaktadır. Edindiğimiz dallar üzerinde çalışmak için checkout komutu ile seçtiğimiz dala yatay geçiş yapmamız gerekmektedir. Şimdi fetch komutunun nasıl kullanıldığını bir örnek üzerinde inceleyelim.

Öncelikle bir yere bağlantısı olmayan yerel bir depo oluşturalım:

```
// Test isimli yeni bir dizin
[oacar@lin repo]$ mkdir test

// Bu dizine geçiş
```



```
[oacar@lin repo]$ cd test/  
  
// Yeni bir depo oluşturma  
[oacar@lin test]$ git init  
  
Initialized empty Git repository in /home/oacar/repo/test/.git/
```

Bu depoyu şimdi GitHub.com üzerindeki başka bir depo ile ilişkilendirelim:

```
[oacar@lin test]$ git remote add test git@github.com:ozcanacar/test.git  
  
[oacar@lin test]$ git remote -v  
test      git@github.com:ozcanacar/test.git (fetch)  
test      git@github.com:ozcanacar/test.git (push)
```

Şimdi fetch komutunu uygulayalım:

```
[oacar@lin test]$ git fetch test  
remote: Counting objects: 3, done.  
remote: Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From github.com:ozcanacar/test  
* [new branch]      branch-a    -> test/branch-a  
* [new branch]      branch-b    -> test/branch-b  
* [new branch]      branch-test -> test/branch-test  
* [new branch]      master      -> test/master
```

Fetch komutu ile github.com/ozcanacar/test.git adresindeki deponun sahip olduğu tüm dalları yerel depomuza çektik. Fetch komutuna parametre olarak verdiğimiz isim remote deponun kayıtlı ismidir. Git remote -v komutuyla ilişki içinde olduğumuz tüm remote depoları görüntüleyebiliriz. Gördüğünüz gibi yerel deponun bir github.com/ozcanacar/test.git klonu bile olması gerekmedi. Bu şekilde birçok depoyu birbiriyle ilişkilendirerek, dalların ve sahip oldukları commitlerin depolar arası transfer edilmesi sağlanmaktadır.

Fetch işleminden sonra dallar yerel depomuza transfer edildi, lakin içinde çalıştığımız dizinde (working copy) hiçbir değişiklik olmadı.

```
[oacar@lin test]$ ls -l  
total 0
```

Diğer depodan edindiğimiz dallara tekrar bir göz atalım:

```
[oacar@lin test]$ git branch -a
```

```

master
remotes/origin/HEAD -> origin/master
remotes/origin/branch-a
remotes/origin/branch-b
remotes/origin/branch-test
remotes/origin/master

```

Merkezi depodan edindiğimiz dallar remotes/origin ibaresine sahiptir. Bu şekilde yerel dallar ile merkezi depodan gelen dalları ayırt etmek kolaylaşmaktadır. Örneğin origin/branch-a bünyesindeki commitlere şu şekilde göz atabiliriz:

```

[oacar@lin test]$ git log --oneline remotes/origin/branch-a
6b82f3d file1 commit
09931cf test1 commit

```

Fetch komutuyla ile edindiğimiz dallardan bir tanesini ana geliştirme dalımız olan master ile şu şekilde birleştirebiliriz (merge):

```

[oacar@lin test]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[oacar@lin test]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 29 13:06 test1

[oacar@lin test]$ git merge origin/branch-a
Updating 09931cf..584a4ec
Fast-forward
 file1 | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1

[oacar@lin test]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 29 13:23 file1
-rw-r----- 1 oacar users 0 Apr 29 13:06 test1

```

Görüldüğü bir master dalı ile origin/branch-a dalları içerik olarak çakışmadıkları için bir fast-forward merge gerçekleşti. Bu şekilde origin/branch-a dalında bulunan tüm dosyaları edinmiş olduk. Merge işleminden sonra master dalının commit geçmişi şu şekilde olacaktır:

```

[oacar@lin test]$ git log --oneline
6b82f3d file1 commit
09931cf test1 commit

```

Şimdi origin/branch-a dalının geçmişine bakalım:

```
[oacar@lin test]$ git log --oneline origin/branch-a
6b82f3d file1 commit
09931cf test1 commit
```

Görüldüğü gibi master ve origin/branch-a dalları aynı commite işaret etmektedirler. Bu şekilde merkezi bir deponun bir dalını kendi çalışmalarımıza eklemiş olduk.

Git Pull

Git pull komutu ile daha önce tanışmıştık. Bağlı olduğumuz bir deponun sahip olduğu değişiklikleri edinmek için pull komutunu kullanabiliriz. Pull komutu fetch ve merge komutlarının birleşiminden oluşmaktadır. Daha önce de belirttiğim gibi, fetch komutu sadece yeni dalları yerel depoya almakta, merge komutu ile bu değişiklikler yerel dallarla birleştirilebilmektedir. Pull komutunu koşturduğumuzda hem fetch hem de merge işlemi arka, arkaya gerçekleştiği için merkezi deponun sahip olduğu tüm değişiklikler içinde olduğumuz yerel dalın bir parçası haline gelmektedirler. Aşağıdaki örnekte master dalında iken pull origin komutunu koşturmaktayız. Bu merkezi deponun master dalı ile yerel deponun master dalını birleştirmektedir. Burada git pull komutu yeterlidir, çünkü yerel depo hangi merkezi depoya bağlı olduğu bilgisine sahiptir.

```
[oacar@lin test]$ git branch
* master

[oacar@lin test]$ git log --oneline
09931cf test1 commit

[oacar@lin test]$ git pull origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
From ssh://github.com:/ozcanacar/test
   09931cf..5eefafb master    -> origin/master
Updating 09931cf..5eefafb
Fast-forward
   xxx | 0
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 xxx
```

Şimdi commit geçmişine bir göz atalım:

```
[oacar@lin test]$ git log --oneline
5eefafb xxx eklendi
09931cf test1 commit
[oacar@lin test]$
```

Ekran çıktısında görüldüğü gibi pull işleminden sonra içinde bulunduğumuz master isimli dala merkezi deponun master isimli dalında yer alan 5eefafb hash değerindeki commit eklenmiştir. Şimdi her iki dalında aynı commite işaret ettiğiniz söyleyebiliriz. Bunun kontrol etmek için merkezi deponun master dalının hangi commite işaret ettiğine şu şekilde bir göz atalım:

```
[oacar@lin test]$ git log --oneline origin/master
5eefafb xxx eklendi
09931cf test1 commit
```

Her iki dal da aynı commite işaret etmektedir, yani pull komutuyla senkronize edilmişlerdir.

Pull işlemi esnasında rebase mekanizmasından faydalananarak, çalıştığımız dal üzerinde yaptığımız değişikliklerin merkezi deponun dalında meydana gelen değişikliklerin üzerine inşa edilmesini sağlayabiliriz. Bu şekilde yerel dalda yaptığımız tüm çalışmaların merkezi dalda meydana gelen değişiklikleri temel almasını sağlayabiliriz. Pull esnasında rebase işlemini şu şekilde gerçekleştirebiliriz:

```
git checkout master
git pull --rebase origin
```

Burada git rebase origin komutunu da kullanmak mümkün. Lakin rebase güncel gelişmeleri merkezi daldan almadığı için rebase işlemi düşündüğümüz şekilde gerçekleşmez. Bu yüzden pull esnasında rebase in yapılması gerekmektedir.

Pull esnasında rebase işlemini standart (default) pull işlemi haline getirmek için, konfigürasyon şu şekilde değiştirilebilir:

```
git config --global branch.autosetuprebase always
git config branch.master.rebase gtrue
```

Bu şekilde bir pull her zaman bir rebase haline gelmektedir. İlk satır yeni oluşturulan dallar, ikinci satır mevcut dallar için rebase işlemini aktiv hale getirmektedir. Bu konfigürasyondan sonra git pull komutunu kullandığınız taktirde, otomatik olarak rebase işlemi gerçekleşecek ve yerel dalda yaptığımız tüm değişiklikler merkezi dalın sahip olduğu değişiklikleri baz alacak şekilde yeniden

yapılandırılacaklardır.

Git Push

Fetch komutunu merkezi deponun sahip olduğu dalları edinmek için kullandık. Bunun tam tersini, yani yerel dallar üzerindeki değişiklikleri merkezi depoya aktarmak için push komutunu kullanabiliriz. Şimdi push komutunun değişik kullanış biçimlerini örnekler üzerinde inceleyelim.

Push komutunun en basit kullanılış şeklinde aşağıda görmekteyiz:

```
[oacar@lin repo]$ git clone ssh://git@github.com/ozcanacar/test.git
Cloning into 'test'...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 10 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (10/10), 935 bytes | 0 bytes/s, done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

Yeni bir klon oluşturduktan sonra, branch listesine göz atıyoruz:

```
[oacar@lin test]$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/branch-a
remotes/origin/branch-b
remotes/origin/branch-test
remotes/origin/master
```

Master dalı üzerinde olduğunu görmekteyiz. Bu dalın ihtiva ettiği dosyalar aşağıda görülmekte:

```
[oacar@lin test]$ ls -l
total 0
-rw-r----- 1 oacar users 0 Apr 30 11:51 test1
-rw-r----- 1 oacar users 0 Apr 30 11:51 xxx
```

Şimdi yeni bir dosya oluşturalım ve master dalına ekleyelim:

```
[oacar@lin test]$ touch abc

[oacar@lin test]$ git add abc
```

```
[oacar@lin test]$ git commit -m "abc eklendi"
[master fac0634] abc eklendi
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 abc
```

Bu değişikliği GitHub.com daki test.git isimli depoya şu şekilde aktarabiliriz:

```
[oacar@lin test]$ git push origin master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 242 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To ssh://git@github.com:ozcanacar/test.git
5eefafb..fac0634 master -> master
```

Master dalında çalışırken yaptığımız değişiklikleri merkezi deponun aynı dalına aktarmak için push komutunun şu şekilde kullanılması yeterlidir:

```
git push
```

Eğer merkezi depoda yeni bir dal oluşturmak istiyorsak, o zaman push komutunu şu şekilde kullanmamız gerekmektedir:

```
git push origin <branch-ismi>
```

Origin bağlı olduğumuz merkezi depoya işaret etmektedir. Bağlı olduğumuz merkezi depo ya da depolar remote listesinde yer almaktadır. Git remote -v komutuyla bu listeye göz atabiliriz.

Push işlemini gerçekleştirebilmek için merkezi deponun seçmiş olduğumuz dalında son yaptığımız pull işleminden sonra değişiklik olmamış olması gerekmektedir. Aksı takdirde push işlemi Git tarafından geri çevrilir. Örneğin bizden önce başka bir kullanıcı tarafından merkezi deponun master dalı için push işlemi gerçekleştirildi ise, bu durumda bizim push işlemimiz Git tarafından uygulanmaz, çünkü mevcut olan değişikliklerin öncelikle pull ile yerel depoya çekilmesi ve birleşim hatalarının (merge conflict) giderilmesi gerekmektedir.

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Git remote komutu ile mevcut iki depoyu veri alışverişi yapacak şekilde ilişkilendirebiliriz.
- Git fetch komutu merkezi depo bünyesinde olan tüm değişiklikleri edinmemizi mümkün kılmaktadır. Pull komutunun aksine fetch komutu otomatik olarak merge işlemi yapmadığından, bu değişiklikleri çalışma alanı dışında incelemek mümkün olmaktadır.
- Git pull komutu ile yerel depo ve çalışma alanı (working copy) merkezi depo ile senkronize edilebilmektedir.
- Git push komutu ile yerel depo bünyesinde meydana gelen değişiklikler merkezi depoya aktarılabilirler.

5. Bölüm

Git İş Akış Modelleri

Üçüncü bölümde dal konsepti ve kullanımı ile tanışmıştık. Bu bölümde feature branch ve git flow workflow olarak bilinen ve dal bazlı yazılım yapmayı mümkün kılan iki modeli tanıtmak istiyorum.

Yazılım projelerinin belli aşamalarında sürüm (release) olarak isimlendirilen çalışır program parçaları oluşturulur. Sürüm oluşturulmadan önce tüm değişikliklerin programcılar tarafında ana yazılım dalı olan master ya da trunk olarak isimlendirilen dala eklenmesi gerekmektedir. Son rötüşlerin yapılabilmesi için ana yazılım dalının bir süre değişikliklere karşı korunması gerekmektedir. Bu işleme yazılımda code freeze yani kod yazma sürecinin dondurulması ismi verilmektedir. Tanımlanan zaman zarfı içinde, bu bir saat, bir gün ya da birkaç gün olabilir, yazılımcıların ana yazılım dalına commit yapmalarına izin verilmez. Uzun süren code freeze devrelerinde yazılımcıların yazdıkları kodları entegre etmeleri zorlaşabilir, çünkü hergün commit görmeyen kod birbirlerinden uzaklaşma eğilimi gösterirler. Tek dallı yapılan yazılım projelerinde ana sıkıntıların başında code freeze gelmektedir.

Bir başka sıkıntı ise, ana dalın (master) sürekli kırılğan bir yapıda olması ve sürüm oluşturmak için yeterli olgunlukta olmamasıdır. Yazılımcılar gün içinde kendi tamamladıkları kod birimlerini diğer yazılımcılara sunabilmek ya da entegre edebilmek için commit yaparlar. Her commit ana dalı kırma potansiyeline sahiptir. Bunun yanı sıra bitmemiş yazılım özellikleri her commit ile ana yazılım dalına aktarılır. Bir sürüm oluşturulduğunda, tamamlanmamış olan yazılım özelliklerinin sürümün bir parçası olma ihtimali yüksektir. Subversion gibi merkezi versiyon yönetim sistemlerinde sürüm alma akışının nasıl işlediğini [bu blog](#) yazımda bulabilirsiniz.

Bahsetmiş olduğum bu deavantajlardan kurtulmanın bir yolu, feature branch olarak isimlendirilen ve sadece bir uygulama özelliğini (feature) ihtiva eden dallar üzerinde çalışmaktan geçmektedir. Her yeni bir uygulama özelliği için yeni bir dal oluşturulur ve bu özelliğin implementasyonu ana kod dalını rahatsız etmeden bu dal bünyesinde gerçekleştirilir. Bu şekilde değişik dallar üzerinde paralel uygulama özellikleri üzerinde çalışmak ve ana kod dalı olan master dalını stabil tutmak mümkündür. Sadece bir özellik tamamlandığında, bu dalın içeriği master dalı ile birleştirilir. Bu işlemin yapılabilmesi için üzerinde çalışılan yan dalın tüm testleri geçmesi ve ana dal ile entegre edilebilir seviyede olması gerekmektedir. Bu şekilde hem ana kod dalı stabil kalacaktır hem de tamamlanmamış olan uygulama özellikleri ana kod dalına aktarılmayacak ve sürüm içinde yer almayacaklardır.

Bu yan dalların başka bir özelliği daha bulunmaktadır. Daha sonra yakından inceleyeceğimiz pull request tekniği ile programcılar arasından yan dallar üzerinden olup, bitenler hakkında fikir alışverişi yapmak kolaylaşmaktadır. Örneğin bir yan dal üzerinde işlemler tamamlandıktan sonra, yan dalı master dalı ile bir araya (merge) getirmeden önce yan dal sahibi bir pull request oluşturarak, bir ekip arkadaşının yan dal üzerindeki değişiklikleri incelemesini talep edebilir. Bu

şekilde kod inceleme seansları yapılabilir ve birlikte keşfedilen hatalar ortadan kaldırılabilir. Pull request tekniği mevcut bir kod birimin ana kod dalının bir parçası haline gelmeden kod hakkında konuşma ve inceleme yapma imkanı sağlamaktadır.

Şimdi feature branch bazlı yazılımın nasıl uygulandığını bir örnek üzerinde inceleyelim. Bu iş akışı (workflow) tarzında oluşturulan dal isimlerinin üzerinde çalışılmak istenen uygulama özelliğini tanımlayıcı ifade gücünde olmaları gerekmektedir. Aksı taktirde paralel bir çok yan dal oluşabileceğinden, bu dalları ayırt etmek güçleşebilir. Aşağıdaki örnekte feature-1 ismini taşıyan bir uygulama özelliğini implemente edeceğiz.

Aşağıda yer alan örnekleri git bash ile Windows işletim sisteminde uyguluyorum. Öncelikle bir yalın (bare) depo oluşturup, bu depoyu klonluyoruz.

```
$ git init --bare myrepo
Initialized empty Git repository in c:/Users/oezcanacar/temp/myrepo/
```

Oluşturduğumuz bu depoyu herhangi bir sunucu üzerinde bulunan bir remote depo olarak düşünebiliriz. Uzakta bulunan bir Git deposuna erişmek için ssh ya da http gibi protokolleri kullanabiliriz. Verdiğim örneklerin uygulanabilir olmasını sağlamak için yerel depoları kullanmayı tercih ettim.

Proje üzerinde çalışabilmek için öncelikle proje deposunu klonlamamız gerekiyor. Bunu şu şekilde yapıyoruz:

```
$ git clone myrepo myrepo-clon
Cloning into 'myrepo-clon'...
warning: You appear to have cloned an empty repository.
done.
```

Boş bir depoyu klonladığımız için "You appear to have cloned an empty repository" uyarısını aldık. Şimdi ilk değişikliği yaparak, master dalını oluşturup, bu değişiklikleri ana depoya aktaralım:

```
$ cd myrepo-clone

$ touch test.txt

$ git add --all
$ git commit -m "initial commit"
[master (root-commit) 9863af5] initial commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test.txt

$ git push origin master
```

```
Counting objects: 3, done.  
Writing objects: 100% (3/3), 215 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To c:/Users/oezcanacar/temp/myrepo  
* [new branch]      master -> master
```

Şimdi hangi dallara sahip olduğumuza bir göz atalım:

```
$ git branch  
* master
```

Bağlı olduğumuz remote depoda hangi dallar mevcut?

```
$ git branch -r  
origin/master
```

Görüldüğü gibi üzerinde çalıştığımız klon depomuz myrepo-clon bünyesinde master isminde bir dal oluştu. Aynı şekilde merkezi depo myrepo bünyesinde de master (origin/master) isminde bir dal mevcut. Bu dalı push esnasında merkezi dal üzerinde oluşturmuş olduk.

Feature branch is akışını uygulayabilmek için bundan sonraki safhalarda master dalını baz alan yan dallar oluşturmamız gerekiyor. Feature-1 ismin taşıyan bir uygulama özelliğini implemente etmek istediğimizi düşünelim. Bu durumda ise bu ismi taşıyan bir yan dal oluşturarak, başlamanız gerekiyor. Bunu şu şekilde yapabiliriz:

```
$ cd myrepo-clon  
  
$ git checkout -b feature-1 master  
Switched to a new branch 'feature-1'  
  
$ git branch  
* feature-1  
master
```

Feature-1 ismini taşıyan yeni bir dal oluşturduk. Checkout komutu ile bu dala geçiş yapmış bulunuyoruz. Bu dal üzerinden yapacağımız tüm değişiklikler sadece bu dalın içeriğini etkileyeceği için master dalı bu değişikliklerden etkilenmeyecektir. Bu şekilde birçok feature branch oluşturarak, paralel değişik uygulama özellikleri üzerinde çalışabiliriz.

Feature-1 dalını yerel depomuzda oluşturduk. Bu çalışmamızı ekip arkadaşlarımızla paylaşmak için oluşturduğumuz yan dalı merkezi depoya aktarmamız gerekiyor. Bu işlemi şu şekilde gerçekleştirebiliriz:

```
$ git push origin feature-1
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 251 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/myrepo
 * [new branch]      feature-1 -> feature-1
```

Şimdi merkezi depomuzda hangi dallar var, ona bir göt atalım:

```
$ git branch -r
origin/feature-1
origin/master
```

Görüldüğü gibi feature-1 ismini taşıyan yan dal merkezi depoya (myrepo) aktarıldı.

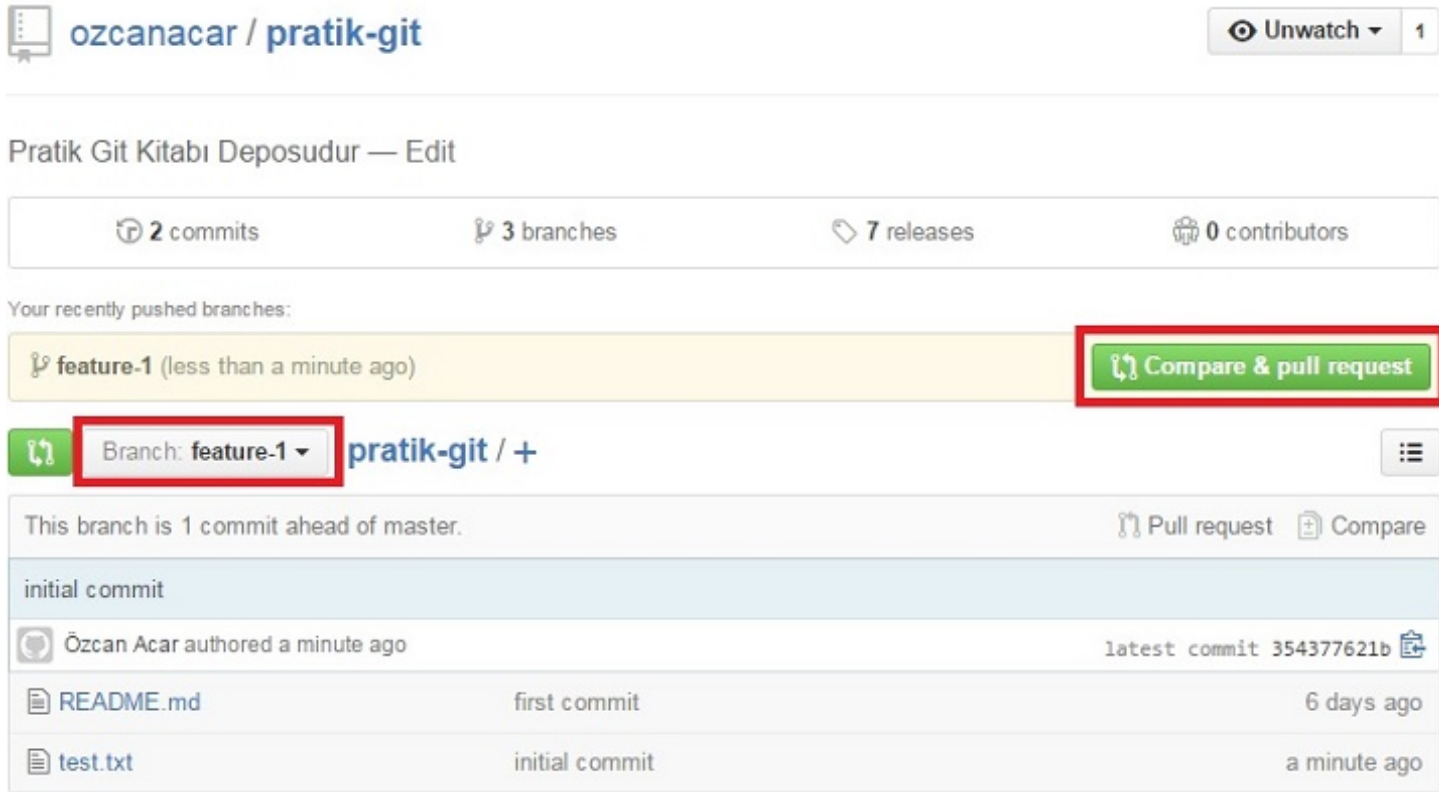
```
$ git remote -v
origin  c:/Users/oezcanacar/temp/myrepo (fetch)
origin  c:/Users/oezcanacar/temp/myrepo (push)
```

Remote komutu merkezi deponun hangi lokasyonda olduğunu göstermektedir. Ekipteki başka bir yazılımcı fetch ya da clone komutlarını kullandığı taktirde, oluşturduğumuz yeni yan dalı edinir. Geriye feature-1 bünyesinde yaptığımız değişiklikleri kontrolü ve master dalı ile birleştirilmesi kalmaktadır. Burada pull request tekniği devreye girmektedir.

Pull request ile istediğimiz şahısları çalıştığımız dal üzerinde inceleme yapmaya davet edebiliriz. Pull request davetini alan şahıslar, bu dal bünyesindeki kodu edinerek, inceleme ve değişiklikler yapabilirler.

Pull request tekniği doğrudan Git bünyesinde yer alan bir özellik değildir. Daha ziyada GitHub ve BitBucket gibi servislerin bünyelerinde barındırdıkları Git depoları için sundukları bir özelliktir. Bunun yanı sıra Atlassian Stash gibi Git bazlı versiyon yönetim servis uygulamalarında da pull request tekniğinin implementasyonlarına rastlamak mümkündür.

Şimdi GitHub bünyesinde bir pull requestin nasıl oluşturulduğunu inceleyelim. Bu amaçla resim 1 de görüldüğü gibi deponun (pratik-git) ana sayfasını seçiyoruz.



Resim 1

Resim 1 de sahip olduğumuz feature-1 dalını görmekteyiz. Bu dalı seçtikten sonra, Compare & pull request butonuna tıklararak, bir pull request oluşturabiliriz.

Bir sonraki resimde pull request için girebileceğimiz mesaj formunu görmekteyiz. Create pull request butonuna tıklayarak, girdiğimiz mesajı ihtiva eden bir pull request oluşturabiliriz. Formun sağ panelinde Assignee isminde bir bölüm yer almaktadır. Bu bölümde pull requestin gideceği şahısları tayin edebiliriz. Örneği basit tutmak için pull requesti kendime gönderiyorum.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master . compare: feature-1 ✓ Able to merge. These branches can be automatically merged.

Feature 1 #3
No description available

Choose a head branch

Branch, tag, commit, or history marker

branch-a
✓ feature-1
master

Write Preview

Leave a comment

Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard.

Create pull request

Labels
None yet

Milestone
No milestone

Assignee
No one—assi

Resim 2

Pull request talepleri genelde ana kod dalı ile birleşme işleminde (merge) son buldukları için, böyle bir talebi oluştururken birleşmenin hangi dal ile yapılması gerektiğini tayin etmemiz gerekiyor. İkinci resmin üst bölümünde base:master, compare:feature-1 seçimlerini görmekteyiz. Feature-1 dalını master dalını baz alarak oluşturdum. Bu yüzden bu dalın birleşeceği dal büyük bir ihtimalle master olacaktır.

Create pull request butonuna tıkladığımda, aşağıda yer alan ve aslında pull request talebini gönderdiğimiz şahsın göreceği sayfaya geçmekteyiz. Bu örnekte talebi alan da, gönderen de ben olduğum için oluşturduğum pull request talebinin inceleyebileceğim sayfaya geçtim. Eğer pull request talebini başka bir şahsa göndermiş olsaydım, bu şahsın göreceği sayfa da resim 3 deki gibi olurdu.

Open ozcanacar wants to merge 2 commits into master from feature-1

Conversation 0 Commits 2 Files changed 2

ozcanacar commented 4 minutes ago Owner

No description provided.

Özcan Acar added some commits 40 minutes ago

- initial commit 3543776
- second commit a17549c

Add more commits by pushing to the **feature-1** branch on ozcanacar/pratik-git.

✓ This branch is up-to-date with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Write Preview Markdown supported Edit in fullscreen

Leave a comment

Attach images by dragging & dropping, selecting them, or pasting from the clipboard.

Close pull request Comment

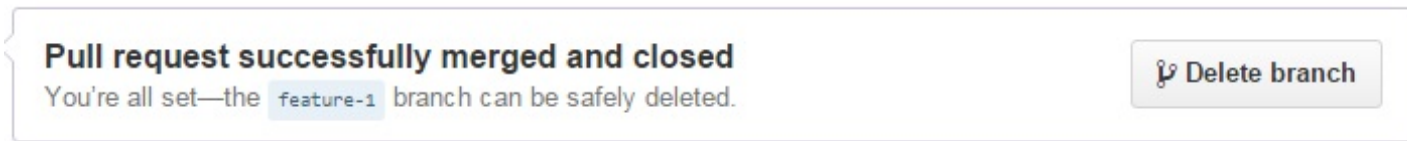
Resim 3

Sayfanın altında yer alan Close pull request butonu ile bu talebi sonlandırabiliriz. Bu durumda pull request talebi hiçbir işlem görmeden silinecektir. Talebi aldığım şahısla ile pull request üzerinden yazışmak için Comment butonu ile bir yorum bırakabiliriz. Resmin üst kısmında "ozcanacar wants to merge 2 commits into master from feature-1" (ozcanacar feature-1 dalında bulunan iki commiti

master dalına aktarmak istiyor)" metnini görmekteyiz. Merge işleminin yönünü daha önce tayin ettik. Birleştirme işlemi feature-1 dalından master dalına doğru gerçekleşecek ve feature-1 dalında yer alan commitler master dalına alınacaktır. Merge işlemi başlatabilmek için Merge pull request butonuna tıklamamız gerekiyor. Akabinde oluşan Confirm merge butonu ile merge işlemi başlatabiliriz.

Merge işleminin neticesini resim 4 de görmekteyiz. Feature-1 dalında yaptığımız tüm değişiklikler bu şekilde master dalına aktarılmış oldu. Bu durumda yapmamız gereken tek bir şey kalıyor: feature-1 dalını silmek. Feature-1 dalının silinmesinde fayda var, çünkü bu dalın var olması için hiçbir sebep kalmadı. Son bulmuş yan dalların temizlenmesi, birçok uygulama özelliğinin paralel geliştirildiği projelerde dal enflasyonunu önlemektedir.

Resim 4 de görüldüğü gibi Delete branch butonuna tıklayarak feature-1 dalını yok edebiliriz.



Resim 4

Git Flow Workflow

Git flow workflow feature branch konseptini baz alan ve dallara belli isimler ve roller atayan bir versiyon kontrol iş akış modelidir. İki model arasında dalların isimlendirilmeleri haricinde bir farklılık olmadığını söyleyebiliriz. Ekip içinde iş akışını kolaylaştırmak için Git flow merkezi bir deponun kullanımını gerekli kılmaktadır. Tüm iş akışı bu depo üzerinden gerçekleştirilir.

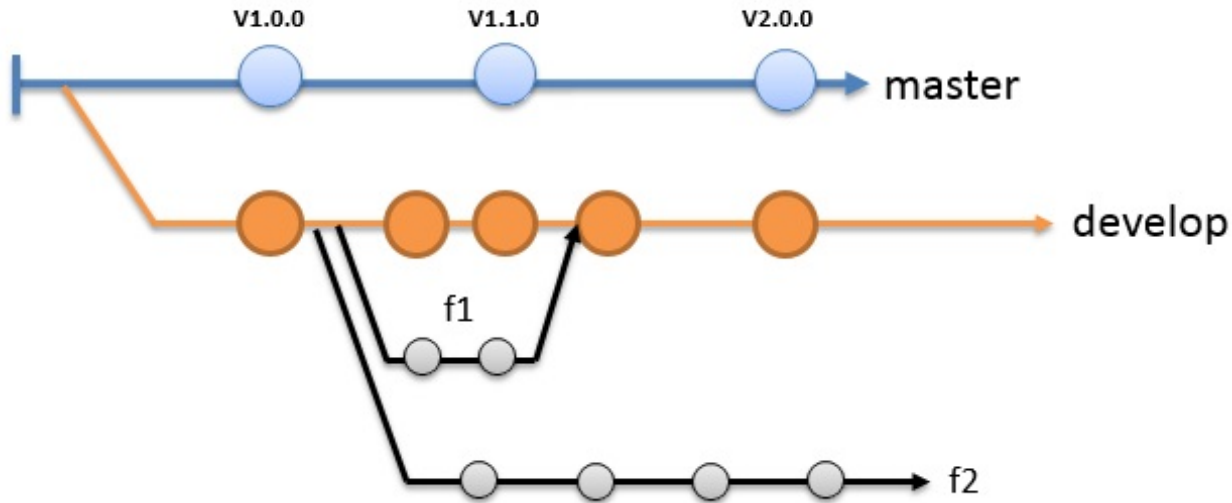
Git flow bünyesinde master dalı yanında develop ismini taşıyan ikinci bir dal daha kullanılır. Master dalı sürüm geçmişini tutarken, develop dalı feature dallarının entegre edildiği dal olarak kullanılır. Feature dalları develop dalını baz alarak oluşturulurlar ve tamamlandıklarında bu dal ile birleştirilirler. Sürüm alma kıvamına gelen develop dalından sürüm dalı oluşturulur ve bu dal gerekli değişikliklerin ardından master dalı ile birleştirilir ve bu birleşme işleminden doğan commit etiketlenerek yeni bir sürüm versiyonu oluşturulur. Develop dalının başlangıç noktasında master dalıdır.

Resim 5 de master ve develop dallarını görmekteyiz. Master dalı 1.0.0, 1.1.0 ve 2.0.0 versiyon numaralı sürümleri oluşturmak için etiketlenmiştir. Bu sürümler oluşturulmadan önce develop dalından sürüm dalları oluşturulmuş, bu dallar master dalı ile birleştirilmiş ve bu birleşme sonucunda oluşan commitler versiyon numaraları ile etiketlenmiştir.



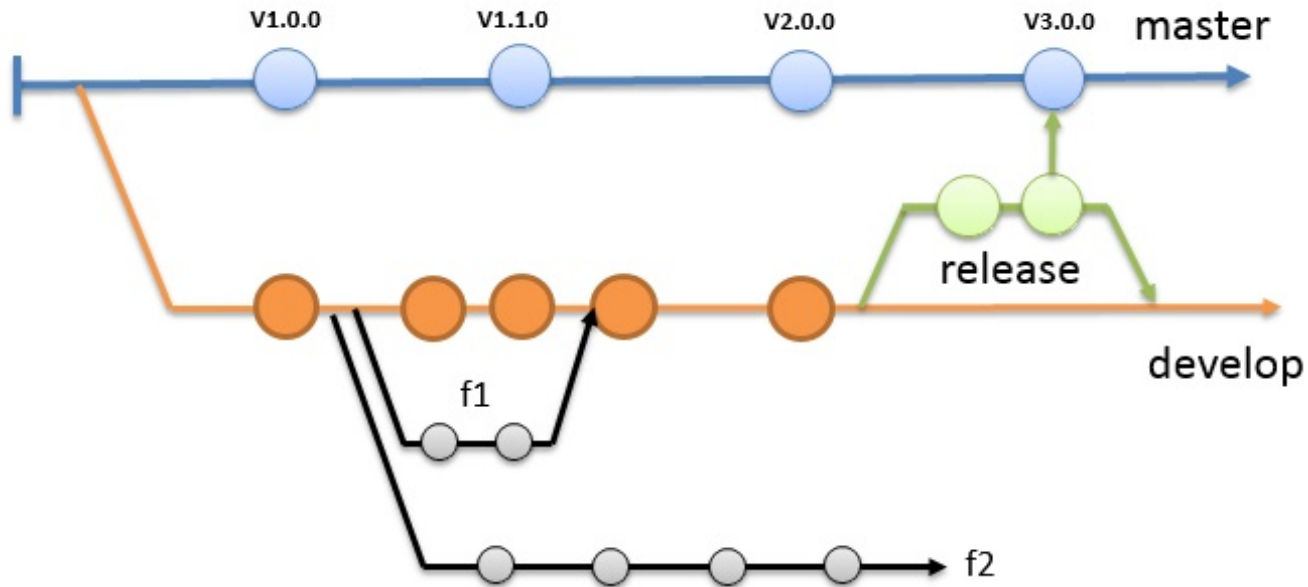
Resim 5

Resim 5 den yola çıkarak feature dalları nasıl oluşturulur, onu inceleyelim. Aşağıdaki resimde develop dalını baz alan f1 ve f2 dallarını görmekteyiz. F1 dalı develop dalındaki birinci commitden sonra oluşturulmuş ve kendi bünyesindeki iki commit gördükten sonra tekrar develop dalı ile birleştirilmiştir. Buna karşın f2 dalı f1 ile aynı kökene sahipken ve dört commit görmesine rağmen, develop dalına paralel olarak hayatını sürdürmektedir. Bu f2 dalında henüz çalışmaların tamamlanmadığı anlamına gelmektedir. F2 bünyesindeki çalışmalar tamamlandığında, içeriği bir merge operasyonu ile develop dalına aktarılacaktır.



Resim 6

Şimdi develop dalında sürüme dahil etmek istediğimiz yeterince uygulama özelliği eklendiğini düşünelim. Bu durumda sürüm oluşturmada önce release branch ismini taşıyan yeni bir dal oluşturmamız gerekiyor. Yani develop dalını master ile birleştirmek yerine, önce develop dalını baz alan yeni bir sürüm (release branch) oluşturuyoruz.

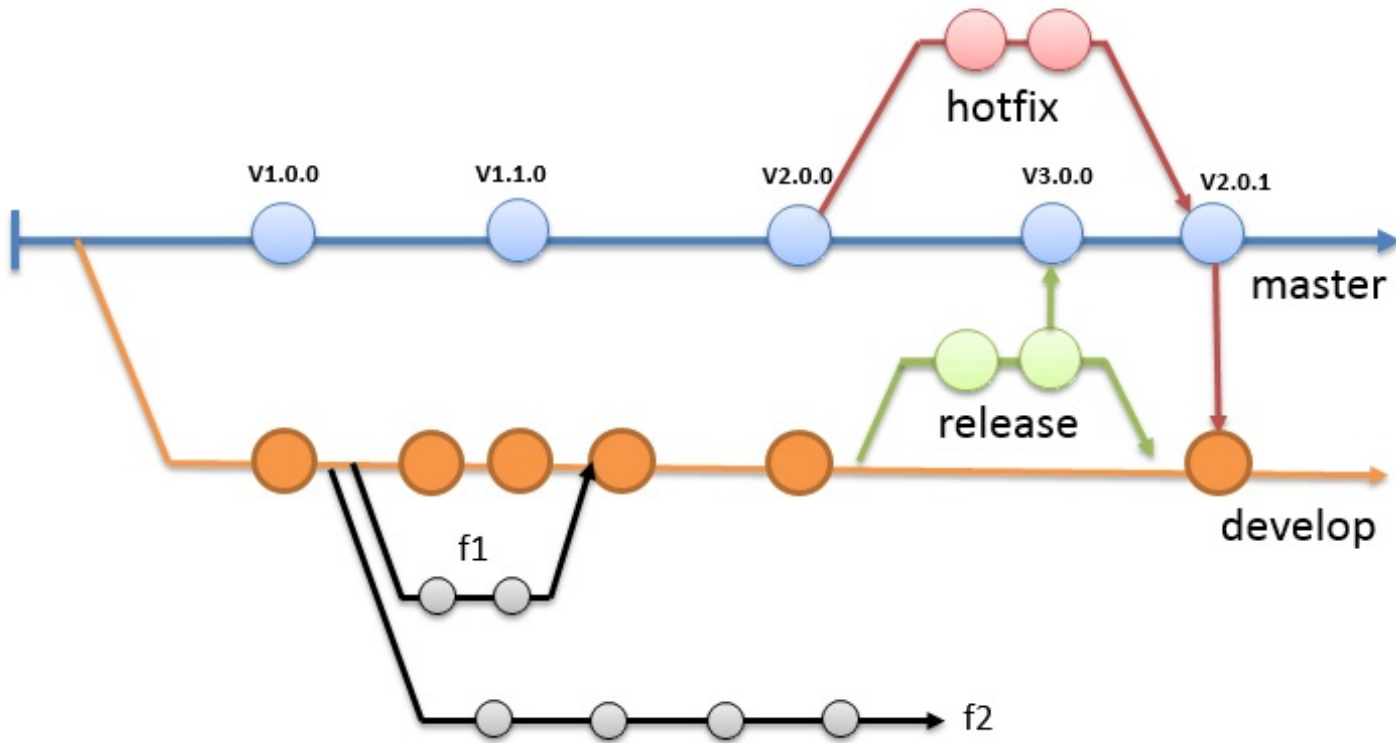


Resim 7

Sürüm dalında kodu sürüm olgunluğuna getirmek için gerekli ayarlar yapıldıktan sonra, bu dal master dalı ile birleştirilir ve master dalı yeni sürüm versiyon numarası ile etiketlenir. Eğer sürüm dalında değişiklikler yapıldı ise, bu dalın develop dalı ile de birleştirilmesi ve yapılan değişikliklerin develop dalına alınması gerekmektedir.

Sürüm dalı oluşturarak, kodu sürüm olgunluğuna getirme işlemlerine paralel olarak, ekip içindeki diğer yazılımcıların bizden bağımsız olarak develop ve feature dallarında kendi işlerine devam etmiş olmalarını sağlamış olmaktadır. Pratikte ekibin diğer bölümü develop ve feature dallarını kullanarak bir sonraki sürüm üzerinde çalışmaya devam etmektedirler. Bu code freeze yapma gerekliliğini ortadan kaldırmaktadır, çünkü sürüm için develop dalını baz alan yeni bir dal oluşturulmuştur ve yazılımcılar kendi işlerine devam edebilmektedirler.

Müşteriye gönderdiğimiz sürümde bir hata oluştuğunu düşünelim. Örneğin 2.0.0 versiyon numarasını sahip olan sürümde müşterimiz bir ya da daha fazla hata keşfetti. Bu durumda bir hotfix dalı oluşturmamız gerekiyor.



Resim 8

Resim 8 de master dalının v2.0.0 etiketini baz alan böyle bir dalın oluşturulduğunu görmekteyiz. Tespit edilen ve 2.0.0 numaralı sürümde bulunan hataların bu hotfix dalında giderildikten sonra master dalına aktarılması ve v2.0.1 versiyon numarasıyla etiketlenmesi gerekmektedir. Aynı şekilde hotfix dalında yapılan değişikliklerin develop dalına aktarılarak, keşfedilen hataların sonraki sürümlerde tekrar ortaya çıkmaları engellenebilir.

Görüldüğü gibi master dalı sadece sürümleri etiketlemek, develop dalı feature dallarında vücut bulan uygulama özelliklerini bünyesinde barındırmak, feature dalları yeni uygulama özellikleri geliştirmek, sürüm dalları yeni sürümler oluşturmadan önce kodu sürüm olgunluğuna getirmek ve hotfix dalları sürümlerde meydana gelen hataları gidermek için kullanılmaktadırlar. İş akışı bu şekilde uygulandığı takdirde ne genel uygulama özellikleri implemente edilirken ne de sürüm oluşturulurken code freeze yapılması gereklidir. Bunun yanı sıra her zaman sürüm alınabilecek stabil bir dal (master) bulunmaktadır.

Git flow iş akışının nasıl işlediğini şimdi bir örnek üzerinde inceleyelim. İşe öncelikle yeni bir yalın depo oluşturarak başlayalım:

```
$ git init --bare git-flow
Initialized empty Git repository in c:/Users/oezcanacar/temp/git-flow/
```

Yeni yalın depomuzu oluşturduk. Şimdi bu depoyu klonluyoruz:

```
$ git clone git-flow git-flow-clone
Cloning into 'git-flow-clon'...
warning: You appear to have cloned an empty repository.
done.
```

Bu depoya geçip, master dalını oluşturuyoruz:

```
// x isimli bir dosya oluşturur
$ touch x

// Bu dosya indexe eklenir
$ git add --all

// Commit işlemi yapılır
$ git commit -m "initial commit" x
[master (root-commit) b3009b0] initial commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 x

// Yerel dalı master dalı olarak merkezi depoya aktarılır
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 210 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
 * [new branch]      master -> master
```

Şimdi develop dalını oluşturuyoruz:

```
$ git checkout -b develop master
Switched to a new branch 'develop'
```

Şimdi f1 ve f2 isimlerindeki ve develop dalını baz alan dalları oluşturuyoruz:

```
$ git checkout -b f1 develop
Switched to a new branch 'f1'

$ git checkout -b f2 develop
Switched to a new branch 'f2'
```

F1 dalında çalışmalarımızı sürdürüyoruz:

```
// f1 isimli daha geçiş
```

```
$ git checkout f1
Switched to branch 'f1'

// xx isimli bir dosya oluşturun
$ touch xx

// Bu dosya indexe eklenir
$ git add --all
$ git commit -m "initial commit"
[f1 70feced] initial commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 xx

// f1 dalı merkezi depoya aktarılır
$ git push origin f1
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 234 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
 * [new branch]      f1 -> f1
```

F1 üzerindeki çalışmalarımızı tamamlıyor ve develop ile birleştiriyoruz:

```
// develop dalına geçiş
$ git checkout develop
Switched to branch 'develop'

// f1 dalı ile birleşim
$ git merge --no-ff f1
Merge made by the 'recursive' strategy.
xx | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 xx
```

--no-ff parametresi kullanıldığında, merge işlemi fast forward türünde olsa bile yeni bir commit oluşturacaktır. Dallar arası birleştirme işlemlerini takip edebilmek için bu tür commitlerin oluşturulmasında fayda vardır. Bunu develop dalının geçmişine baktığımızda, görmekteyiz:

```
$ git checkout f1
Switched to branch 'f1'

$ git log --oneline
70feced initial commit
b3009b0 initial commit
```

```
$ git checkout develop

$ git log --oneline
33cd1d0 Merge branch 'f1' into develop
70feced initial commit
b3009b0 initial commit
```

Develop dalındaki ve 33cd1d0 hash değerindeki commit develop ile f1 dallarının birleştirilmeleri esnasında oluştu. Birleşme işlemi gerçekleştiği için artık f1 dalını silebiliriz:

```
$ git checkout develop
Switched to branch 'develop'

$ git branch -d f1
Deleted branch f1 (was 70feced).
```

Şimdi develop dalını da merkezi depoya aktaralım:

```
$ git push origin develop
Counting objects: 1, done.
Writing objects: 100% (1/1), 234 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
 * [new branch]      develop -> develop
```

Buraya kadar neler yaptık, kısa bir özet yapalım:

- Merkezi yalın depoyu oluşturduk.
- Bu deponun bir klonunu oluşturduk.
- Master dalını oluşturduk.
- Master dalını baz alan develop dalını oluşturduk.
- Develop dalını baz alan f1 ve f2 feature dallarını oluşturduk.
- F1 dalındaki çalışmalarımızı tamamlayıp, içeriğini develop dalı ile birleştirdikten sonra bu dalı sildik.

Şimdi yeni bir sürüm alalım. Ama amaçla develop dalını baz alan bir sürüm dalı oluşturuyoruz:

```
$ git checkout -b release-v1.0.0 develop
Switched to a new branch 'release-v1.0.0'
```

Bu daldaki iken örneğin sürüm numarasını v1.0.0 olarak değiştirebiliriz. Version numaraları genelde version.properties gibi bir dosya içinde tutulur. Bu dosyada bir önceki sürümün numarası

yer almaktadır ve bu sebepten dolayı bu içeriğin sürüm dalında yeni sürüm numarasını yansıtacak şekilde adapte edilmesi gerekmektedir. İlk sürümde olduğumuz için öncelikle bu dosyayı oluşturup, sürüm dalına ekliyoruz:

```
$ touch release.properties

$ echo v1.0.0 > release.properties

$ git add --all

$ git commit -m "sürüm dosyası oluşturuldu"
[release-v1.0.0 58bf4a0] sürüm dosyası oluşturuldu
1 file changed, 1 insertion(+)
create mode 100644 release.properties

$ git push origin release-v1.0.0
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 307 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
 * [new branch]      release-v1.0.0 -> release-v1.0.0
```

Push komutu ile sürüm dalını merkezi depoya aktarmış olduk. Oradaki dal yapısı şimdi bu şekilde olacaktır:

```
$ git branch -r
origin/develop
origin/fl
origin/master
origin/release-v1.0.0
```

F1 dalını yerel depomuzdan silmiştik. Aynı şekilde merkezi depomuzdan da silmemiz gerekmektedir. Bu işlemi şu şekilde yapabiliriz:

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git push origin :fl
To c:/Users/oezcanacar/temp/git-flow/
 - [deleted]          fl
```


Şimdi sürüm dalında yaptığımız değişiklikleri master dalına aktaralım ve bu dalı sonlandıralım:

```
$ git checkout master

$ git merge --no-ff release-v1.0.0
Merge made by the 'recursive' strategy.
 release.properties | 1 +
xx                  | 0
2 files changed, 1 insertion(+)
create mode 100644 release.properties
create mode 100644 xx
```

Şimdi master dalını sürüm versiyon numarası ile etiketliyoruz:

```
// v1.0.0 isimli etiketi oluşturur
$ git tag -a v1.0.0

// Mevcut etiket listesini gösterir
$ git tag
v1.0.0
```

Sürüm dalında yaptığımız tüm değişiklikleri bu şekilde master dalına almış ve orada yeni bir sürüm etiketi oluşturmuş olduk. Artık bu etiket ismini kullanarak, bu sürümü master dalından almamız mümkündür.

Sürüm dalında yaptığımız değişiklikleri aynı zamanda develop dalına da almamız gerekmektedir. Bu işlemi şu şekilde gerçekleştirebiliriz:

```
// develop dalına geçiş
$ git checkout develop
Switched to branch 'develop'

// Sürüm dalı ile birleştirme
$ git merge --no-ff release-v1.0.0
Merge made by the 'recursive' strategy.
 release.properties | 1 +
1 file changed, 1 insertion(+)
create mode 100644 release.properties
```

Şimdi release dalını silebiliriz:

```
$ git branch -d release-v1.0.0
Deleted branch release-v1.0.0 (was 58bf4a0).
```

Şimdi v1.0.0 sürümünde bir hata olduğu haberini aldık ve hemen bir hotfix dalı oluşturuyoruz. Bu dalın başlangıç noktası master dalında oluşturduğumuz v1.0.0 etiketidir.

```
$ git checkout -b hotfix-v1.0.1 tags/v1.0.0
Switched to a new branch 'hotfix-v1.0.1'
```

Sürüm numaraları için [buradaki](#) blog yazıma bakabilirsiniz. Şimdi hotfix-v1.0.1 ismini taşıyan dal üzerindeyiz ve v1.0.0 etiketi ile etiketlenmiş sürümün içinde bulunan bilimum dosyaya sahip durumdayız. Hatayı gideriyoruz ve versiyon numarasını değiştiriyoruz:

```
$ echo 1.0.1 > release.properties

$ git add --all

$ git commit -m "hotfix version 1.0.1 created"
[hotfix-v1.0.1 ec32714] hotfix version 1.0.1 created
 2 files changed, 2 insertions(+), 1 deletion(-)

$ git push origin hotfix-v1.0.1
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 519 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
 88c5da6..d924886 develop -> develop
 9923d55..715ce4f master -> master
* [new branch]      hotfix-v1.0.1 -> hotfix-v1.0.1
```

Hatayı giderdik ve şimdi hotfix dalını master dalı ile birleştiriyoruz:

```
$ git checkout master

$ git merge hotfix-v1.0.1
Merge made by the 'recursive' strategy.
 release.properties | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 release.properties
```

Git yerel master dalında yaptığımız değişiklikleri şimdi merkezi master dalına aktarıyoruz:

```
$ git push
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (2/2), 317 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To c:/Users/oezcanacar/temp/git-flow/
    33cd1d0..88c5da6  develop -> develop
    b3009b0..9923d55  master -> master
```

Buraya kadar neler yaptık, kısaca özetleyelim:

- Yeni bir sürüm almak için develop dalını baz alan ve release-v1.0.0 ismini taşıyan yeni bir dal oluşturduk.
- Sürüm dalında yaptığımız değişiklikleri merkezi depoya aktardık.
- F1 dalını merkezi depodan sildik.
- Sürüm dalındaki değişiklikleri master dalına aktardık ve v1.0.0 etiketini oluşturduk.
- Sürüm dalındaki değişiklikleri develop dalına aldık.
- Sürüm dalını yerel depodan sildik.
- 1.0.0 sürümünde hata olduğunu öğrendik ve master dalındaki v1.0.0 etiketini baz alan yeni bir hotfix dalı oluşturduk.
- Hotfix dalı bünyesinde hatayı giderdik.

Şimdi hotfix dalındaki değişiklikleri master ve develop dalına alarak, bu dalı sonlandırabiliriz:

```
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

$ git branch
  develop
  f2
  hotfix-v1.0.1
* master

$ git merge --no-ff hotfix-v1.0.1
Merge made by the 'recursive' strategy.
 release.properties | 2 +-
    xx                | 1 +
 2 files changed, 2 insertions(+), 1 deletion(-)
```

Şimdi yeni bir sürüm etiketi oluşturuyoruz:

```
$ git tag -a v1.0.1
```

Hotfix dalındaki değişiklikleri develop dalına alıyoruz:

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff hotfix-v1.0.1
Merge made by the 'recursive' strategy.
 release.properties | 2 +-
 xx                  | 1 +
 2 files changed, 2 insertions(+), 1 deletion(-)
```

Ve en son işlem olarak hotfix dalını yok ediyoruz:

```
$ git push origin --delete hotfix-v1.0.1
To c:/Users/oezcanacar/temp/git-flow/
- [deleted]          hotfix-v1.0.1
```

En son komut ile merkezi depodaki hotfix dalını sildik. Aynı şekilde bu dalı yere depodan da şu şekilde silebiliriz:

```
$ git branch -d hotfix-v1.0.1
Deleted branch hotfix-v1.0.1 (was ec32714).
```

Şimdi yerel ve merkezi depo dal listemize bir göz atalım:

```
$ git branch
 develop
  f2
* master

$ git branch -r
 origin/develop
 origin/master
 origin/release-v1.0.0
```

Yerel depomuzda master, develop ve f2 depoları, merkezi depoda master, develop ve release-v1.0.0 dalları mevcut. Merkezi depoda bulunan origin/release-v1.0.0 dalını silmemiz gerekiyor, çünkü bu daldaki değişiklikleri daha önce master dalına aktardık. Bu sürüm dalını şu şekilde silebiliriz:

```
$ git push origin --delete release-v1.0.0
To c:/Users/oezcanacar/temp/git-flow/
- [deleted]          release-v1.0.0
```

Şimdi master ve develop dallarının geçmişine bir göz atalım. Master dalı geçmişi şu şekilde:

```
$ git checkout master
```

```
$ git log --oneline
715ce4f Merge branch 'hotfix-v1.0.1'
ec32714 hotfix version 1.0.1 created
9923d55 Merge branch 'release-v1.0.0'
58bf4a0 sürüm dosyası oluşturuldu
33cd1d0 Merge branch 'f1' into develop
70feced initial commit
b3009b0 initial commit
```

Deponun etiket geçmişini şu şekilde edinebiliriz:

```
$ git tag -l
v1.0.1
v1.0.0
```

Peki hangi etiket hangi commit hash değerine işaret etmektedir. Bunu şu şekilde bulabiliriz:

```
$ git rev-list v1.0.0 | head -n 1
9923d5507f8009c867d5baca4ca67a91f9876ea1

$ git rev-list v1.0.1 | head -n 1
715ce4f33349107c1cb3c947cc09d7994e4a2c66
```

Master dalındaki herhangi bir etiketin altında sürümü edinmek için şu şekilde ilerleyebiliriz:

```
$ git checkout master

$ git log --oneline
715ce4f Merge branch 'hotfix-v1.0.1'
ec32714 hotfix version 1.0.1 created
9923d55 Merge branch 'release-v1.0.0'
58bf4a0 sürüm dosyası oluşturuldu
33cd1d0 Merge branch 'f1' into develop
70feced initial commit
b3009b0 initial commit

$ git checkout tags/v1.0.0
Previous HEAD position was 715ce4f... Merge branch 'hotfix-v1.0.1'
HEAD is now at 9923d55... Merge branch 'release-v1.0.0'

$ git log --oneline
9923d55 Merge branch 'release-v1.0.0'
58bf4a0 sürüm dosyası oluşturuldu
33cd1d0 Merge branch 'f1' into develop
70feced initial commit
```

```
b3009b0 initial commit
```

Git checkout tags/v1.0.0 ile v1.0.0 etiketinin altındaki commite ulaşmış ve checkout işlemini gerçekleştirmiş olduk. Şimdi sahip olduğumuz durum v1.0.0 sürümünün kopyasıdır. Bu noktadan itibaren örneğin yeni bir hotfix sürümü oluşturabiliriz.

First Parent Geçmişi

Aşağıdaki örnekte görüldüğü gibi bir merge esnasında oluşan commit nesnesinin iki parent referansı vardır:

```
$ git log --merges

commit 787868d34349234ksd9083d94k
Merge: ed19i8 jku84fr4
Author: ...
```

Merge commit nesnesinde yer alan ed19i8 hash değeri first parent olarak tanımlanmaktadır. Bu merge esnasında HEAD in işaret ettiği commit nesnesidir. Bu şekilde nerede merge işleminin gerçekleştiğini görmekteyiz.

Git flow bünyesinde uygulama özelliklerinin implemente edildiği feature dallarının develop bünyesinde entegre edildiğini gördük. Develop dalının geçmişine baktığımızda, entegre edilen feature dalları bünyesinde yapılan commitleri de görürüz. Bu zaman içinde çok uzun bir liste haline gelebilir. Bunun yerine sadece bir feature dalının develop dalına entegre edilirken oluşan merge commit listesini görebilseydik, develop dalına hangi feature dallarının entegre edildiğini yani hangi uygulama özelliklerinin develop dalında bulunduğunu görebilirdik. Bunu first parent geçmişi üzerinden görebiliriz.

First parent yardımıyla bir daldaki feature entegrasyon listesini şu şekilde edinebiliriz:

```
$ git log --first-parent --oneline develop

7faz654 Merge branch 'Feature-4 implemented'
9iju76z Merge branch 'Feature-3 implemented'
0o87hzt Merge branch 'Feature-2 implemented'
hj789ki Merge branch 'Feature-1 implemented'
```

First parent yardımıyla aldığımız liste develop dalının sıkıştırılmış (compact) halidir. Feature dallarındaki commit nesnelerini görmek zorunda kalmadan, doğrudan develop dalının feature entegrasyon geçmişini görebilmekteyiz.

Herşey de olduğu gibi burada da bir istisna yok değil. Bir feature dalının develop dalı ile birleştirilmesi esnasında fast-forward-merge olmuş ise, bu durumda feature dalının beraberinde getirdiği tüm commitler first parent geçmişinde yer alacaktır, çünkü onların işaret ettikleri first parent değişmemiştir ve develop dalındaki HEAD ile aynıdır. Bunun yanı sıra develop dalı bünyesinde merge işlemlerinin yapılması, first parent geçmişinin de feature bazında değil, tüm değişiklikleri ihtiva edecek şekilde oluşmasına sebep verecektir. Bu sebeple develop dalında sadece feature dal entegrasyonu için merge işlemleri yapılmalıdır.

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Feature branch konsepti uygulama özelliklerini birbirlerinden bağımsız bir şekilde geliştirmek için kullanılan bir Git iş akış modelidir.
- Git flow workflow feature branch konseptini baz alan ve dallara belli isimler ve roller atayan bir versiyon kontrol iş akış modelidir.
- First parent yardımıyla bir dalda hangi uygulama özelliklerinin entegre edildiğini öğrenmek mümkündür.

6. Bölüm

Git Nesne Veri Tabanı

Bu bölüme kadar kullandığımız branch, checkout, clone, rebase gibi komutlar Git terminolojisinde porselen (porcelain) olarak isimlendirilen üst katmana ait komutlardır. Bu komutlar yardımıyla depoda olup, bitenleri bilmeden, daha soyut bir seviyede Git ile çalışmak mümkündür.

Birde bunun yanı sıra Git'in nasıl çalıştığını gösteren ve alt katmana ait olan komutlar mevcuttur. Bu tür komutlara altyapı ya da tesisat anlamına gelen plumbing ismi verilmektedir. Çok alt seviyede işlem yapmayı mümkün kılan bu komutlar aracılığı ile bir Git deposunun yapısını ve çalışma tarzını inceleyebiliriz. Bu bölümde bunu yapacağız.

Normal şartlar altında günlük işlerimizde alt seviye komutlarını kullanma ihtiyacı duymayız. Ama Git'in nasıl işlediğiniz daha iyi kavrayabilmek için bu alt seviye komutların nasıl kullanıldığını örnekler üzerinde göstermek istiyorum.

İlk inceleyeceğimiz commit cat-file ismini taşıyor. Aşağıda yer alan örnekte 715ce4f hash değerindeki commit yapısının nasıl olduğunu görmekteyiz:

```
$ git log --oneline
58bf4a0 sürüm dosyasi oluşturuldu
33cd1d0 Merge branch 'f1' into develop
70feced initial commit
b3009b0 initial commit

$ git cat-file commit 58bf4a0
tree a022a9af74479265719a8d91742ee44af3c6de98
parent 33cd1d0b564fa730cba96890c2812c9ebc91092e
author Özcan Acar <oezcanacar@xxx> 1439820883 +0200
committer Özcan Acar <oezcanacar@xxx> 1439820883 +0200

"sürüm dosyasi oluşturuldu"
```

Cat-file komutuna verdiğimiz commit parametresi ile bir commit nesnesinin içeriğine bakmak istediğimiz belirtmiş olduk. Yukarıda yer alan örnekte cat-file commit komutu bize 715ce4f hash değerine sahip olan commit nesnesinin hangi parçalardan oluştuğunu göstermektedir. Resim 1 de bu commit nesnesinin yapısını görmekteyiz. Görüldüğü gibi commit nesnesi tree, parent, author ve committer gibi bilgileri içermektedir. Peki bu bilgiler ne ifade etmektedir?

Git add komutu ile üzerinde değişiklik yaptığımız dosya ve dizinleri depoya eklerken, yapılan değişikliklerin bir nevi fotoğrafını çekmiş (shapshot) oluyoruz. Bu resmi commit nesnesinde yer alan tree (ağaç) nesneleri temsil etmektedir. Bir commit bünyesinde hangi dizin ve dosyaların yer aldığını görmek için bu tree nesnesine göz atabiliriz:

```
$ git cat-file tree a022a9af74479265719a8d91742ee44af3c6de98
```

```
100644 release.properties \u00c4KwóDS²s«C†93100644 x æ9d ÑÖCK8b)
```

Cat-file komutu ile ne yazık ki bir tree nesnesini içeriğine bakmamız mümkün değildir, çünkü tree nesneleri Git bünyesinde binary blob şeklinde tutulmaktadırlar. Ls-tree komutu ile tree nesnelerinin içeriğine şu şekilde bakabiliriz:

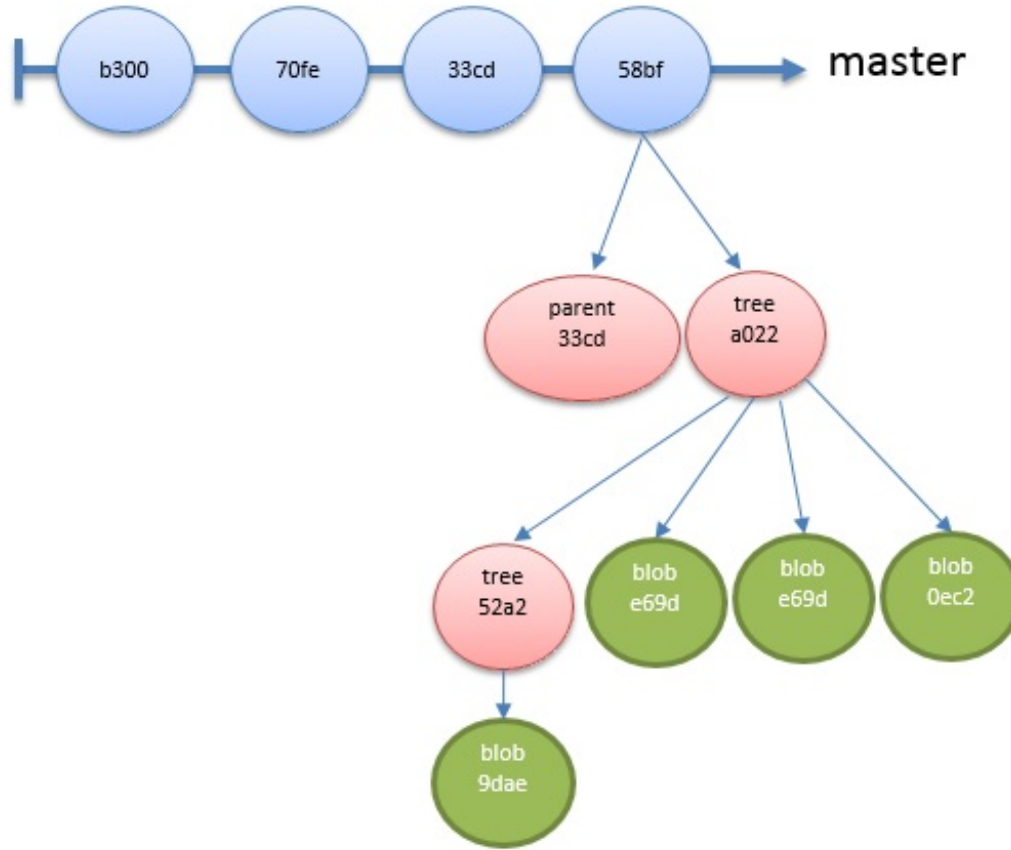
```
$ git ls-tree a022a9af74479265719a8d91742ee44af3c6de98
100644 blob 0ec25f7505c14bad77f34453b2730417ab431293    release.properties
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    test1.txt
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    test2.text
100644 tree 52a266a58f2c028ad7de4dfd3a72fdf76b0d4e24    test-dizini
```

Görüldüğü gibi ls-tree komutu bize commit esnasında depoya eklediğimiz dosya ve dizinleri göstermektedir. Blob ibaresi ile dosyaları, tree ile dizinleri görmekteyiz. Bir tree nesnesindeki bir dosyanın içeriğine bakmak için cat-file komutunu şu şekilde kullanabiliriz:

```
$ git cat-file blob 0ec25f7505c14bad77f34453b2730417ab431293
```

Aynı şekilde ls-tree komutu ile tree nesnesi içinde başka bir tree nesnesine göz atabiliriz:

```
$ git ls-tree e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```



Resim 1

Örneklerde görüldüğü gibi Git commit esnasında depoya eklemek istediğimiz dosya ve dizinleri bir tree nesnesi olarak yapılandırmaktadır. Bu şekilde iç, içe geçmiş dizin ve ve dosyaları depoda tutmak mümkün hale gelmektedir.

Resim 1 e tekrar baktığımızda her commit nesnesinin parent ibaresi ile kendinden önceki commit nesnesine işaret ettiğini görmekteyiz. Bu şekilde bir commit nesnesinden bir önceki commit nesnesine geçmek mümkün hale gelmektedir. Dal üzerindeki ilk commitde parent ibaresi bulunmaz.

Sizce Git bir dalı bir depo bünyesinde nasıl tutuyor? Bakalım:

```
$ git cat-file -t master
commit
```

Cat-file komutu ve -t parametresi ile master dalının yapısına bir göz attığımızda, commit değerini aldık. Bu depo bünyesinde dalların yapılan en son commite işaret ettiğini göstermekte. Şimdi bu commitin içeriğine bir göz atalım:

```
$ git cat-file commit master
tree 5805b676e247eb9a8046ad0c4d249cd2fb2513df
author Özcan Acar <oezcanacar@xxx> 1439909784 +0200
committer Özcan Acar <oezcanacar@xxx> 1439909784 +0200

"bu bir commit mesajı"
```

Master dalında yapılan en son commit nesnesi 5805b6 hash değerine sahip bir tree nesnesinden oluşmaktadır. Bu tree nesnesi bünyesinde commit esnasında depoya eklediğimiz dosya ve dizinleri ihtiva etmektedir. Aynı bilgiyi şu şekilde de edinebiliriz:

```
$ git cat-file commit HEAD
tree 5805b676e247eb9a8046ad0c4d249cd2fb2513df
author Özcan Acar <oezcanacar@xxx> 1439909784 +0200
committer Özcan Acar <oezcanacar@xxx> 1439909784 +0200

"bu bir commit mesajı"
```

HEAD referansını daha sonra inceleyeceğiz.

Dalların işaret ettikleri en son commitler `.git/refs/heads/` dizininde yer alan dosyalarda tutulmaktadır. Bu dizinde her dal için o dalın ismini taşıyan bir dosya bulunmaktadır ve bu dosya dalın işaret ettiği en son commit hash değerini ihtiva etmektedir. Örneğin master dalının işaret ettiği commit nesnesini şu şekilde edinebiliriz:

```
$ cat .git/refs/heads/master
48ca6b97707f4e4f1e94edd7744a0b06454aeae4
```

Master dalının işaret ettiği en son commit 48ca6b9 hash değerindeki commit nesnesidir. Master dalının geçmişine `log` komutu ile baktığımızda, 48ca6b9 commitin en üstte olduğunu görmekteyiz:

```
$ git log --oneline
48ca6b9 bu bir commit mesajı
95ca634 ikinci commit
45ca3b6 initial commit
```

Görüldüğü gibi master bünyesinde yapılan en son commit 48ca6b9 değerine sahiptir ve bu commit nesnesinin içeriğini `cat-file commit HEAD` ile edinebiliriz:

```
$ git cat-file commit HEAD
tree 5805b676e247eb9a8046ad0c4d249cd2fb2513df
author Özcan Acar <oezcanacar@xxx> 1439909784 +0200
committer Özcan Acar <oezcanacar@xxx> 1439909784 +0200
```

```
"bu bir commit mesajı"
```

Peki HEAD tam olarak nedir ve neye işaret etmektedir? HEAD .git dizinde bulunan bir dosyadır ve içeriği normal şartlarda bir dala işaret eder. Bu dal checkout ile yan geçiş yaptığımız ve üzerinde çalıştığımız daldır. Şimdi HEAD dosyasının içeriğine bir göz atalım:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Görüldüğü gibi HEAD dosyasının içinde bir satır bulunmaktadır. Bu satır HEAD nesnesinin hangi dala işaret ettiğini yani hangi dalın checkout ile aktif hale geldiğini göstermektedir. Yukarıdaki örnekte HEAD master dalına işaret etmektedir. Şimdi yeni bir dal oluşturalım ve checkout ile bu dala geçelim. Akabinde HEAD dosyasının içeriğini inceleyelim:

```
$ git checkout -b develop master
Switched to a new branch 'develop'

$ cat .git/HEAD
ref: refs/heads/develop
```

Görüldüğü gibi HEAD şimdi checkout ile oluşturduğumuz ve kullanıma aldığımız dala işaret etmektedir. Aşağıdaki komutu girdiğimizde, Git önce HEAD dosyası aracılığı ile kullanılan dalı, akabinde bu dalın işaret ettiği en son commiti .git/refs/heads/master dosyasında bulmaktadır.

```
$ git checkout master

$ git cat-file commit HEAD
tree 5805b676e247eb9a8046ad0c4d249cd2fb2513df
author Özcan Acar <oezcanacar@xxx> 1439909784 +0200
committer Özcan Acar <oezcanacar@xxx> 1439909784 +0200

"bu bir commit mesajı"
```

Hatırlarsanız üçüncü bölümde "You are in 'detached HEAD' state" hatası ile karşılaşmıştık. Eğer HEAD dosyası bir dal değil de bir commit nesnesine işaret ediyorsa, bu checkout komutu ile o commit nesnesine geri dönmüş anlamına gelmektedir. Bu durumda Git yeni yapılan commitleri nereye koyması gerektiğini bilemediği için "You are in 'detached HEAD' state" hatası ile karşılaşmaktayız.

Şimdi detached HEAD durumuna nasıl düşebileceğimizi bir örnek üzerinde inceleyelim. Öncelikle

master dalındaki en son dan bir önceki commit nesnesini checkout yapıyoruz:

```
$ git log --oneline
c91b602 ikinci commit
4c46ca5 ilk commit

$ git checkout 4c46ca5
Note: checking out '4c46ca5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at 4c46ca5... ilk commit
```

Görüldüğü gibi master dalındaki ilk commite geçtiğimiz andan itibaren detached HEAD oluştu, çünkü HEAD dosyasının içeriği artık şu şekilde:

```
$ cat .git/HEAD
4c46ca58ab4c3e89a2008385091e0c974846119a
```

Bu durumda iken yeni bir commit yaptığımızı düşünelim:

```
$ touch abc

$ git add --all

$ git commit -m "test"
[detached HEAD 4a05c50] test
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 abc
```

Bu durumda HEAD dosyası bu yeni commite işaret edecektir:

```
$ cat .git/HEAD
4a05c5076c74b1252fa98f4df106163a2f9f3ac3

$ git log --oneline
4a05c50 test
```



```
4c46ca5 ilk commit
```

Tekrar master dalının işaret ettiği en son commite geçmek istediğimizde:

```
$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

    4a05c50 test

If you want to keep them by creating a new branch, this may be a good time
to do so with:

    git branch new_branch_name 4a05c50

Switched to branch 'master'
```

Görüldüğü gibi detached HEAD modundayken yeni commit nesneleri oluşturabilmekteyiz, lakin dalın güncel pozisyonuna geri döndüğümüzde yukarıda yer alan mesajı almaktayız. Bu durumda yaptığımız commiti ya unutacağız ya da "git branch newbranchname 4a05c50" ile bu commit nesnesini başlangıç olarak alan yeni bir dal oluşturacağız. Bu şekilde oluşturduğumuz commiti gözle görülür bir yere koymuş olmaktadır. Öyle ya da böyle oluşturduğumuz yeni commit 4a05c50 depoda bir yerlerde duruyor ve 4a05c50 hash değerini kullanarak istediğimiz zaman bu commite geri dönebiliriz. Burada yapılabilecek en mantıklı işlem detached HEAD pozisyonuna düşmemek, düşülse bile yeni commitler oluşturmamak, yeni commitler oluştursak bile, bu commitler için yeni dallar oluşturmak olacaktır.

Git Nesne Veri Tabanı

Git commit, tree ve blob gibi nesneleri kendine has bir nesne veri tabanında tutmaktadır. Bu bölümde bu veri tabanının nasıl bir yapıya sahip olduğunu yakından inceleyeceğiz.

Git'in nesne veri tabanı .git/objects dizininde yer almaktadır. Şimdi bu dizinin ve dolaylı olarak nesne veri tabanının yapısına bir göz atalım:

```
$ cd .git/objects/
$ ls -l
total 0
4a
4c
58
```

```
75
7e
c9
e6
f5
info
pack
```

Git yönettiği her nesneyi bir dosya olarak tutar. Bu dosyanın ismi nesnenin türüne göre değişik verilerden hesaplanmış SHA-1 anahtarındır (checksum). Git bu dosyaları hesapladığı SHA-1 anahtarının ilk iki karakterinden oluşan dizinlerde tutar. Örneğin 4a05c5076c74b1252fa98f4df106163a2f9f3ac3 SHA-1 anahtarına sahip bir dosya 4a isimli dizinde yer alır. 4a dizinine baktığımızda, bu dosyayı 05c5076c74b1252fa98f4df106163a2f9f3ac3 isminde görürüz. Dizin ismi ile dosya ismi bir araya getirildiğinde, dosyanın gerçek ismi ortaya çıkmış olmaktadır.

Daha önce kullandığımız cat-file komutu ile böyle bir dosyanın yani bir Git nesnesinin içeriğine bir göz atabiliriz. Öncelikle dosyanın hangi tür Git nesnesi olduğunu keşfetmemiz gerekiyor:

```
$ git cat-file -t 4a05c5076c74b1252fa98f4df106163a2f9f3ac3
commit
```

Bu dosyanın bir commit nesnesi olduğunu öğrenmiş olduk. Şimdi içeriğine bakabiliriz:

```
$ git cat-file commit 4a05c5076c74b1252fa98f4df106163a2f9f3ac3
tree 7e49477ad9b766be1c44e80f365bb8c75162d0d5
parent 4c46ca58ab4c3e89a2008385091e0c974846119a
author Özcan Acar <oezcanacar@xxx> 1439974497 +0200
committer Özcan Acar <oezcanacar@xxx> 1439974497 +0200

test
```

Git'in nesne veri tabanında kendi dosyalarımızı da tutabiliriz. Örneğin şimdi yeni bir dosya oluşturalım ve bu dosyayı nesne veri tabanına ekleyelim:

```
$ echo "test" > test.txt

$ git hash-object -w test.txt
9daeafb9864cf43055ae93beb0afd6c7d144bfa4
```

Hash-object komutu ile oluşturduğumuz test.txt dosyası için yeni bir SHA-1 değeri oluşturabiliriz. Bu örnekte -w parametresini kullandığımız için test.txt dosyası içinde bulunduğumuz deponun

nesne veri tabanına eklendi. Bu dosyanın içeriğine şu şekilde ulaşabiliriz:

```
$ git cat-file blob 9daeafb9864cf43055ae93beb0afd6c7d144bfa4
test
```

Şimdi test.txt dosyasının içeriğini değiştirelim ve dosyayı tekrar Git'in nesne veri tabanına ekleyelim:

```
$ echo "bir satır" >> test.txt

$ git hash-object -w test.txt
eab5bbfd64a57c6d1e2f02d545f44e11474ca8

$ echo "yeni bir satır" >> test.txt

$ git hash-object -w test.txt
f8d24ae725ff1ea5c789e757a57c29e5b4164b1e
```

Bu işlemler ardından test.txt dosyasının üç değişik sürümü deponun nesne veri tabanında şu şekilde yer alacaktır:

```
$ find .git/objects -type f
.git/objects/9d/aeafb9864cf43055ae93beb0afd6c7d144bfa4
.git/objects/ea/b5bbfd64a57c6d1e2f02d545f44e11474ca8
.git/objects/f8/d24ae725ff1ea5c789e757a57c29e5b4164b1e
```

Şimdi test.txt dosyasının değişik sürümlerine bir göz atalım:

```
$ git cat-file -t 9daeafb9864cf43055ae93beb0afd6c7d144bfa4
blob

$ git cat-file blob 9daeafb9864cf43055ae93beb0afd6c7d144bfa4
test

$ git cat-file blob eab5bbfd64a57c6d1e2f02d545f44e11474ca8
test
bir satır

$ git cat-file blob f8d24ae725ff1ea5c789e757a57c29e5b4164b1e
test
bir satır
yeni bir satır
```

Sadece dosyaları değil, içlerindeki diğer dizin ve dosyalarla dizinleri de deponun nesne veri

tabanında tutmak mümkün. Örneğin aşağıdaki dizin ve dosya yapımızı nesne veri tabanına eklemek istediğimizi düşünelim:

```
dizin1
|
|__ dosya1
|
|__ dizin2
    |
    |__ dosya2
```

Bu dizin ve ihtiva ettikleri dosyaları ihtiva eden bir Git tree nesnesi oluşturabilmek için bu yapıyı Git'in indexine eklememiz gerekiyor. Daha önceki örneklerde commit öncesi değişikliğe uğrayan dosyaları git add komutu ile indexe eklemiştik. Burada add komutu yerine alt katman komutlarından olan update-index komutundan faydalanacağız. Bu örnekteki amacımız commit için hazırlık yapmak değil, bir tree nesnesinin nasıl oluşturulduğunu ve nesne veri tabanına nasıl eklendiğini göstermektir.

Önce oluşturduğumuz yeni dizin ve dosyaları indexe ekliyoruz. Bu bir nevi git add işlemidir.

```
$ git update-index --add dizin1/dizin2/dosya1

$ git update-index --add dizin1/dosya1

Changes to be committed:
$ git status
On branch develop
(use "git reset HEAD <file>..." to unstage)

    new file:   dizin1/dizin2/dosya1
    new file:   dizin1/dosya1
```

Dizin ve dosyaları indexe ekledik. Şimdi bu dizin yapısını ihtiva eden bir tree nesnesi oluşturuyoruz:

```
$ git write-tree
c5e8b78bcc6fe836365ac19a66eda9255e507637
```

Görüldüğü gibi Git indexe eklediğimiz dizin ve dosyaları ihtiva eden bir tree nesnesi oluşturdu. Bu tree nesnesinin SHA-1 anahtarı yani hash değeri c5e8b78bcc6fe836365ac19a66eda9255e507637 şeklindedir. Bu nesnenin içeriğini şu şekilde inceleyebiliriz:

```
$ git cat-file -t c5e8b78bcc6fe836365ac19a66eda9255e507637
```

```
tree

$ git ls-tree c5e8b7
040000 tree 4ecaea18f8bde474f758361b00136b09d85a0d7d    dizin1

$ git cat-file -t 4ecaea18f
tree
```

Git tarafından oluşturulan tree nesnesi kendi bünyesinde başka bir tree nesnesi ihtiva etmektedir. Onun içeriğine şu şekilde ulaşabiliriz:

```
$ git ls-tree 4ecaea18f8
040000 tree 02caca1b9b2d3dffb592aa5138cf3607a6ab38f3    dizin2
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    dosya1
```

Bu örnekte görüldüğü gibi Git dosyaları blob, dizinleri de tree nesnesi olarak nesne veri tabanında tutmaktadır.

Şimdi alt katman komutları yardımıyla bir commit nesnesini nasıl oluşturabileceğimizi yakından inceleyelim. Bir önceki örnekte c5e8b7 hash değerini taşıyan bir tree nesnesi oluşturduk. Bu tree nesnesini kullanarak, yeni bir commit nesnesi oluşturabiliriz. Yeni bir commit oluşturmadan önce master dalındaki en son commitin hangisi olduğunu öğrenmemizde fayda var, çünkü en son commiti oluşturacağımız yeni commitin parent commiti olarak tanımlamamız gerekmektedir. Master dalındaki en son commiti şu şekilde öğrenebiliriz:

```
$ git log --oneline -n 1
c91b602 ikinci commit
```

Şimdi bu iki değeri yani tree ve parent tree nesnesini kullanarak yeni bir commit nesnesi oluşturabiliriz:

```
$ git commit-tree c5e8b78 -p c91b602
Bu yeni bir commit
^Z
a3cda0159d6399964b5790a6444fc9d5bb513cb2
```

Bu örnekte görüldüğü gibi commit-tree komutuna oluşturduğumuz tree nesnesinin hash değeri olan c5e8b78 değerini verdikten sonra -p parametresi ile parent nesnesinin hash değerini tanımlıyoruz. Return tuşuna bastığımızda, Git bizden bir commit mesajı girmemiz için bekleyecektir. Yeni commit mesajını girdikten ve return tuşuna bastıktan sonra Windows işletim sistemlerinde CRTL+Z, Unix işletim sistemlerinde CTRL+D ile giriş işlemlerini tamamlayabiliriz.

Akabinde ekranda gördüğümüz yeni commit nesnesinin hash değeridir. Böylece deponun nesne veri tabanına kendi oluşturduğumuz izin ve dosyalardan oluşan yeni bir commit nesnesi eklemiş olduk.

Görüldüğü gibi alt katman komutları ile tree ve commit nesneleri oluşturmak mümkün. Bu nesneleri doğrudan deponun nesne veri tabanına ekleyebiliyoruz. Lakin bir Git deposuyla verimli olarak çalışabilmek için üst katman komutlarını kullanmamızda fayda var. Üst katman komutları alt katmanda bulunan komutlar yardımıyla bir versiyon kontrol sistemi için gerekli işlemlerin yapılmasını mümkün kılmaktadırlar.

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Git komutları üst seviye ve alt seviye olarak iki gruba ayrılmaktadır. Üst seviye komutları ile (clone, checkout, pull, push vs.) bir versiyon kontrolü için gerekli işlemler yapılabilirken, alt seviye komutları ile (cat-file, ls-tree, update-index vs.) Git'in nesne veri tabanında işlem yapılabilmektedir. Üst seviye komutları alt seviye komutlarını kullanarak, nesne veri tabanı üzerinde işlem yapabilmektedirler.
- Git bir depo bünyesinde meydana gelen değişiklikleri tree, commit ve blob nesneleri olarak bir nesne veri tabanında tutmaktadır.
- Cat-file komutu ile Git nesnelerinin içeriğini incelemek mümkündür.
- Ls-tree komutu ile tree nesnesinin içeriğine göz atmak için kullanılabilir.
- HEAD kullanılan dala işaret eden bir göstergeç dosyadır.
- Git update-index komutu ile yeni ve değişikliğe uğrayan dosyalar indexe (staging area) eklenebilir.
- Write-tree komutu ile index bünyesindeki dosyalar nesne veri tabanına aktarılır.

7. Bölüm

Git Pratik Bilgiler

Git Stash

Daha önceki bölümlerde belirttiğim gibi klonlanan bir Git deposu iki bölümden oluşmaktadır. Bunlar:

- Depo (repository)
- Depoda yer alan dosyaların güncel kopyaları (working copy)

Klonladığımız herhangi bir depoya baktığımızda, bu iki bölümü görebiliriz. Aşağıdaki örnekte deponun kendisi .git dizininde yer almaktadır. README.md, test.txt ve test2.txt dosyaları depoda yer alan versiyonlanmış dosyaların kopyalarıdır. Bu dosyalar Git terminolojisinde working copy ya da working tree olarak isimlendirilirler. Depoda bulunan dosyalar üzerinde işlem yapabilmek için böyle bir working copy ile çalışmamız gerekiyor. Mevcut bir depoyu klonladıktan sonra böyle bir working copy sahibi oluruz.

```
$ ls -la
total 5
drwxr-xr-x  6 oezcanac Administ  0 Aug 17 10:51 .
drwxr-xr-x 10 oezcanac Administ 4096 Aug 19 08:51 ..
drwxr-xr-x 13 oezcanac Administ 4096 Aug 17 10:52 .git
-rw-r--r--  1 oezcanac Administ 12 Aug 17 10:40 README.md
-rw-r--r--  1 oezcanac Administ  0 Aug 17 10:40 test.txt
-rw-r--r--  1 oezcanac Administ  0 Aug 17 10:51 test2.txt
```

Üzerinde çalıştığımız dala göre working copy içeriği değişik olacaktır. Working copy bünyesindeki yer alan ve Git'in kontrolünde olan bu dosyalar üzerinde değişiklik yapabilir ve yeni dosyalar ekleyebiliriz. Git status komutu bize working copy üzerinde yapılan değişiklikleri göstermektedir. Git add komutu ile yaptığımız değişiklikleri Git indexe (staging area) ekleyebilir ve akabinde commit ile bu değişiklikleri depoya aktarabiliriz.

Working copy içindeki dosyalar üzerinde yaptığımız her değişiklik working copy nin kirlenmesine sebep olmaktadır. Bu durumda iken başka bir dala geçmemiz sorunlu olabilir. Başka bir dala geçmeden yaptığımız değişiklikler için bir commit yapmak istemiyor da olabiliriz. Değişik sebeplerden dolayı bir dal üzerinde yaptığımız değişiklikleri kaybolmamaları adına koruma altına almamız gerekebilir. Bu gibi durumlarda yapılan değişiklikleri stash komutuyla rafa kaldırmak ve üzerinde çalıştığımız durumu dondurmamız mümkündür. Rafta olan bu değişiklikleri istediğimiz zaman geri alarak, kaldığımız yerden devam edebiliriz.

Şimdi bu durumu bir örnek üzerinde inceleyelim. Aşağıdaki örnekte master dalındayım ve test.txt dosyasının içeriğini değiştirip, test3.txt isminde yeni bir dosya oluşturuyorum.

```

$ ls -l
total 1
-rw-r--r--  1 oezcanac Administ  12 Aug 17 10:40 README.md
-rw-r--r--  1 oezcanac Administ   0 Aug 20 14:01 test.txt
-rw-r--r--  1 oezcanac Administ   0 Aug 20 14:01 test2.txt

$ echo "test" >> test.txt

$ touch test3.txt

$ 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:   test.txt

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

        test3.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

Git status komutu working copy nin kirlilik derecesini göstermektedir. Bu durumda iken develop dalına geçmek istediğimizi düşünelim:

```

$ git checkout develop
error: Your local changes to the following files would be overwritten by checkout:
        test.txt
Please, commit your changes or stash them before you can switch branches.
Aborting

```

Eğer Git develop dalına geçmemize izin verseydi, test.txt dosyasında yaptığımız değişiklikleri kaybedecektik, çünkü Git develop dalında bulunan test.txt dosyasını working copy alanına kopyalayacaktı. Bu durumda test.txt dosyasında yaptığımız değişiklikleri kaybetmiş olacaktık. Yeni oluşturduğumuz test3.txt dosyası için bir sıkıntı yok, çünkü develop dalında böyle bir dosya bulunmuyor. Lakin develop dalına geçtiğimizde bu dosya working copy içinde varlığını sürdürecektir ve bu kafa karıştırıcı bir durum olabilir. Master dalındayken yaptığımız tüm değişiklikleri, buna indexin içeriği de dahil rafa şu şekilde kaldırabilir ve temiz bir working copy ile develop dalına geçebiliriz:

```
$ git add test3.txt

$ git stash
Saved working directory and index state WIP on master: a17549c second commit
HEAD is now at a17549c second commit

$ git status
On branch master
nothing to commit, working directory clean
```

Stash komutu hem mevcut dosyalar üzerinde yaptığımız değişiklikleri hem de indexi rafa kaldırmaktadır. Yeni oluşturduğumuz test3.txt dosyasını da rafa alabilmek için bu dosyayı önce add komutu ile indexe eklememiz gerekiyor. Akabinde stash komutu ile yaptığımız tüm değişiklikleri rafa kaldırıyoruz. Stash işleminden sonra verdiğim örnekte görüldüğü gibi status komutu bize çalıştığımız working copy nin temiz olduğunu yani hiçbir değişiklik görmediğini söylemektedir.

Çalıştığımız dizine tekrar göz attığımızda, stash işleminden sonra test3.txt dosyasının bu dizinde artık yer almadığını görmekteyiz. Stash komutu bu dosyayı fiziksel olarak rafa kaldırmıştır. Aksi taktirde status komutu böyle bir dosyanın mevcudiyetini bize bildirirdi.

Şimdi develop dalına şu şekilde geçiş yapabiliriz:

```
$ git checkout develop
Switched to branch 'develop'
```

Görüldüğü gibi dosyalar üzerinde herhangi bir değişiklik yapmamışcasına develop dalına geçebildik, ama rafa kaldırılan dosyalar bir yerlerde Git tarafından saklanıyor. Rafın içeriğine şu şekilde göz atabiliriz:

```
$ git stash list
stash@{0}: WIP on master: a17549c second commit
```

Şimdi master dalına geri dönelim ve raftaki değişiklikleri nasıl geri alabileceğimizi inceleyelim.

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ ls -l
total 1
-rw-r--r-- 1 oezcanac Administ 12 Aug 17 10:40 README.md
```

```
-rw-r--r--    1 oezcanac Administ      0 Aug 20 14:41 test.txt
-rw-r--r--    1 oezcanac Administ      0 Aug 20 14:41 test2.txt
```

Şimdi stash apply komutu ile raftaki değişiklikleri geriye alabiliriz:

```
$ git stash apply
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   test3.txt

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:   test.txt
```

Görüldüğü gibi stash apply rafta bulunan en son stash kaydını alarak, içindeki değişiklikleri working copy ye uyguladı. Şimdi içinde bulunduğumuz dizin yapısı şöyle:

```
$ ls -l
total 1
-rw-r--r--    1 oezcanac Administ    12 Aug 17 10:40 README.md
-rw-r--r--    1 oezcanac Administ      5 Aug 20 14:41 test.txt
-rw-r--r--    1 oezcanac Administ      0 Aug 20 14:41 test2.txt
-rw-r--r--    1 oezcanac Administ      0 Aug 20 14:41 test3.txt
```

Tekrar rafın yapısına bir göz atalım:

```
$ git stash list
stash@{0}: WIP on master: a17549c second commit
```

Stash apply ile değişiklikleri rafdan almış olsak bile, stash komutuyla oluşturduğumuz kayıt hala mevcut. Bu kaydı şu şekilde silebiliriz:

```
$ git stash drop
Dropped refs/stash@{0} (5accbed863491e8969cdda2f6b1a24cf1d8ffe93)

$ git stash list
```

Stash drop komutuyla raf kaydını silmiş olduk. Git bünyesinde stash bir stack olarak implemente edilmiştir. Stash alanında aşağıdaki örnekte görüldüğü gibi birden fazla kayıt olabilir yani birden

fazla stash işlemi gerçekleştirmiş olabiliriz:

```
$ git stash list
stash@{0}: WIP on master: a17549c second commit
stash@{1}: WIP on master: 7876g5f ilk commit
stash@{2}: WIP on master: 89wj823 initial commit
```

Eğer stash apply komutuna bir parametre vermezsek, rafın en tepesinde olan stash kaydı (stash@{0}) uygulanır. Kaydın numarasını kullanarak belli bir stash yapısına şu şekilde geri dönebiliriz:

```
$ git stash apply stash@{2}
```

Stash apply komutu stash kaydını bilmez. Stash drop komutu ile rafın en üstündeki kaydı silebiliriz. Stash drop komutu ile belli bir kayda sahip stash kaydını da silmek şu şekilde mümkündür:

```
$ git stash drop stash@{2}
```

İlk yaptığımız stash apply işleminde şöyle bir sonuç almıştık:

```
$ git stash apply
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   test3.txt

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:   test.txt
```

Peki neden test.txt dosyası index içinde değil de, yeni bir dosya olarak görünüyor. Hatırlarsanız bu dosyayı stash işlemi öncesi git add ile indexe eklemiştik. Stash apply esnasında indexin yapısını da geriye almak istiyorsak, --index parametresini şu şekilde kullanmamız gerekmektedir:

```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   test.txt
```

```
new file:   test3.txt
```

Peki rafda bulunan değişiklikleri baz alan yeni bir branch nasıl oluşturabiliriz? Şu şekilde:

```
$ git stash branch feature-2
M       test.txt
A       test3.txt
Switched to a new branch 'feature-2'
On branch feature-2
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       modified:   test.txt
       new file:   test3.txt

Dropped refs/stash@{0} (0a99ecef014672c7dd7f7391c16c3cba5fe6c6e4)

$ ls -l
total 1
-rw-r--r--  1 oezcanac Administ      12 Aug 17 10:40 README.md
-rw-r--r--  1 oezcanac Administ       5 Aug 20 14:56 test.txt
-rw-r--r--  1 oezcanac Administ      0 Aug 20 14:41 test2.txt
-rw-r--r--  1 oezcanac Administ      0 Aug 20 14:56 test3.txt
```

Stash listesinin en tepesinde yer alan kaydı baz alan feature-2 isminde yeni bir dal oluşturmuş olduk.

İnteraktif Staging

Commit işleminden önce değişiklik yapılan ve yeni oluşturulan dosyaların index ismi verilen staging area bölümüne eklenmesi gerektiğini daha önceki örneklerimizde incelemistik. Sadece index içinde bulunan dosyalar commit esnasında depoya aktarılmaktadırlar.

İndexe dosya ekleme işlemini add komutu ile gerçekleştirmekteyiz. Aşağıdaki örnek tüm değişiklikleri indexe eklemektedir.

```
$ git add --all
```

Herhangi bir dosyayı indexe şu şekilde ekleyebiliriz:

```
$ git add <dosya-ismi>
```

İndexe dosya ekleme işlemini interaktif olarak da gerçekleştirebiliriz. Bu genellikle yapılan birçok değişikliği değişik commit nesnelerinde gruplamak için kullanılan bir yöntemdir. Aşağıda yer alan örnekte oluşturduğumuz dört yeni dosyayı iki commit için hazırlayacağız.

Öncelikle oluşturduğumuz dosyalara bir göz atalım:

```
$ ls -l
total 0
-rw-r--r--  1 oezcanac Administ      0 Aug 21 13:26 test1.txt
-rw-r--r--  1 oezcanac Administ      0 Aug 21 13:26 test2.txt
-rw-r--r--  1 oezcanac Administ      0 Aug 21 13:26 test3.txt
-rw-r--r--  1 oezcanac Administ      0 Aug 21 13:26 test4.txt

$ git status
On branch master

Initial commit

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

    test1.txt
    test2.txt
    test3.txt
    test4.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Şimdi interaktif staging oturumunu başlatalım:

```
$ git add -i

        staged      unstaged path

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch      6: diff        7: quit        8: help
What now>4
```

test1.txt ve test2.txt dosyalarını indexe eklemek için 4 (add untracked) rakamını giriyoruz:

```
1: test1.txt
2: test2.txt
3: test3.txt
4: test4.txt
```



```
Add untracked>> 1,2
* 1: test1.txt
* 2: test2.txt
  3: test3.txt
  4: test4.txt
Add untracked>>
added 2 paths

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch       6: diff        7: quit        8: help
What now>
```

Dört rakamını girdikten sonra Git bize Add untracked>> bölümünde hangi dosyaları indexe eklemek üzere seçmek istediğimizi sormaktadır. Birinci ve ikinci dosyaları seçtikten sonra ana menüye geri dönüyoruz. Şimdi 1 seçeneği ile devam edelim. Bu bize indexe eklediğimiz ve commit için bekleyen dosyaları gösterecektir.

```
What now>1
      staged      unstaged path
  1:      +0/-0      nothing test1.txt
  2:      +0/-0      nothing test2.txt

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch       6: diff        7: quit        8: help
```

Şimdi q tuşu ile çıkıp, commit işlemini gerçekleştirebiliriz:

```
$ git commit -m "initial commit"
[master (root-commit) 35e12db] initial commit
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test1.txt
create mode 100644 test2.txt
```

Bu şekilde test1.txt ve test2.txt dosyalarını ihtiva eden yeni bir commit nesnesi oluşturmuş olduk. Geriye test3.txt ve test4.txt dosyaları kalmış oldu:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test3.txt
```

```
test4.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Şimdi tekrar interaktif modda iken neler yapabileceğimize bir göz atalım:

```
$ git add -i
                staged      unstaged path

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch       6: diff        7: quit        8: help
What now>
```

- status: İndexin yapısını gösterir.
- update: Değişikliğe uğrayan dosyaların indexe eklenmesini mümkün kılar.
- revert: İndexe eklenen dosyaların geri alınması sağlar.
- add untracked: Yeni dosyaların indexe eklenmesini sağlar.
- patch: Yapılan tüm değişikliklerin index ekleme işlemi öncesinde tekrar gözden geçirilmesi ve seçilmesini mümkün kılar.
- diff: İndex ile son commit arasındaki farklılıkları gösterir.
- quit: İnteraktif oturumu sonlandırır.
- help: Yardım sayfasına yönlendirir.

Şimdi bir dosya üzerinde birden fazla değişiklik yapmak istediğimizi ve bu değişikliklerden sadece bazılarını indexe eklemek istediğimizi düşünelim. Bu durumda yoldaşımız `--patch` parametresi olacaktır. Aşağıdaki örnekte `test1.txt` dosyası üzerinde iki değişiklik yapıyoruz. Bu değişiklikler öncesinde dosyanın içeriğine bir göz atalım:

```
$ cat test1
test
test
test
```

Şimdi içeriği şu şekilde değiştiriyoruz:

```
$ cat test1
test
test
test
dördüncü satır
```

```
beşinci satır
```

Şimdi patch ekleme işlemini başlatalım:

```
$ git add --patch
diff --git a/test1.txt b/test1.txt
index 0867e73..4ac2260 100644
--- a/test1.txt
+++ b/test1.txt
@@ -1,3 +1,5 @@
 test
 test
 test
+dördüncü satır
+beşinci satır
Stage this hunk [y,n,q,a,d,/,e,]?
```

Git burada ilk olarak test1.txt dosyasının en son commit (HEAD) nesnesindeki hali ile mevcut halini diff komutu ile kıyasladı ve bu dosyaya son iki satırın yeni eklendiğini keşfetti. En son satırda Git'in bizden bir giriş beklediğini görmekteyiz. Burada ? işaretini girerek, Git'den yardım alabiliriz:

```
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,3 +1,5 @@
 test
 test
 test
+dördüncü satır
+beşinci satır
Stage this hunk [y,n,q,a,d,/,e,]?
```

Örneğin sadece dördüncü satırın test1.txt dosyasında indexe eklenmesini istiyorsak, bu durumda e tuşu ile devam edebiliriz. Bir sonraki adımda diff işleminden doğan içerik editör içinde açılacaktır.

```
# Manual hunk edit mode -- see bottom for a quick guide
@@ -1,3 +1,5 @@
test
test
test
+dördüncü satır
+beşinci satır
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging. If it does not apply cleanly, you will be given
# an opportunity to edit again. If all lines of the hunk are removed,
# then the edit is aborted and the hunk is left unchanged.
```

Diff işlemi bize dosya içindeki yeni satırları + işareti ile göstermektedir. İndexe almak istemediğimiz satırları - işaretiyle tanımlayabiliriz. Editörden çıktığımız zaman, bu dosyanın indexe eklenmesi işlemi tayin ettiğimiz satırlar ile gerçekleşecektir. Bu şekilde indexede ve dolaylı olarak bir sonraki commitde hangi değişikliklerin yer alacağını kendimiz tayin edebilmekteyiz.

Pratik Bilgi ve İşlemler

Çalışma alanımızdaki değişikliklerin listesini status komutu ile alabildiğimizi daha önce gördük. Şimdi bu çıktının kısaltılmış haline bir göz atalım:

```
$ git status --short
?? test1
?? test2
?? test3
M x
```

M modified, ?? ile yeni dosya anlamına gelmektedir.

İndex (staging area) içinde neler olduğunu merak edenler:

```
$ git diff --staged
diff --git a/test1 b/test1
```

```
new file mode 100644
index 00000000..e69de29
```

Bu diff index ile deponun en son commiti olan HEAD arasındaki farklılık göstermektedir. Bu örnekte test1 isimli dosyayı git add test1 ile indexe ekledim.

İndexe yaptığım eklentileri nasıl geri alabilirim?

```
$ git reset HEAD .
```

Burada tüm dal olarak HEAD yani son commite geri döndük ve aynı zamanda index silindi.

Git bünyesinde aynı içeriğine sahip olan dosyalar aynı hash değerine sahiptirler. Görelim:

```
$ cp test1 test1-1

$ git hash-object test1
test1      test1-1

$ git hash-object test1
9daeafb9864cf43055ae93beb0afd6c7d144bfa4

$ git hash-object test1-1
9daeafb9864cf43055ae93beb0afd6c7d144bfa4
```

Git'in yönettiği iki nesnenin aynı hash anahtarına sahip olması, içeriklerinin aynı olduğu anlamına gelmektedir. Bu yüzden iki değişik dosyanın aynı hash anahtarına sahip olması tehlikeli bir durumdur. Buna Git terminolojisinde hash kolizyonu (çarpışması) ismi verilmektedir. Ama hemen üzölmek için bir sebep yok. 2^{160} (iki üzeri 160) değişik kombinasyonda bir hash çarpışmasının oluşma ihtimali çok düşük. Linux kernel Git deposunda oluşan nesnelerin sahip oldukları hash havuzu 2^{21} büyüklüğünde. Gerisi malum :)

Peki commit hash değerlerine nasıl ulaşabilirim? Reflog ile:

```
$ git reflog
500e859 HEAD@{0}: checkout: moving from test to 500e859f2a766510374cf916b8a4f4af
6cbc728 HEAD@{1}: commit: dosya eklendi
ae72a0b HEAD@{2}: checkout: moving from 500e859f2a766510374cf916b8a4f4afb7b1b7de
500e859 HEAD@{3}: checkout: moving from master to 500e859f2a766510374cf916b8a4f4
ae72a0b HEAD@{4}: clone: from https://github.com/ozcanacar/backend.git
```

Şu şekilde bir commite geri dönmek mümkün:

```
$ git checkout HEAD@{3}
A      test1
HEAD is now at 500e859... test1 dosyasi eklendi.
```

Etiketlerin sahip oldukları ya da işaret ettikleri commitlerin sahip oldukları hash anahtarları merak edenler:

```
$ git show-ref --dereference --tags
...
500e859f2a766510374cf916b8a4f4afb7b1b7de refs/tags/v1.5
```

Hangi etiketlerin olduğuna bakmak istersek:

```
$ git tag -l v1.*

v1.5      1.5 sürümü
v1.4      1.4 sürümü
v1.3      1.3 sürümü
v1.1      1.1 sürümü
```

Aşağıdaki örnekte her commit yanında ait oldukları etiket ve dal gösterilmektedir:

```
$ git log --oneline --decorate

500e859 (HEAD, tag: v1.5) test2.txt eklendi.
6cbc728 (tag: v1.1) test1.txt eklendi.
```

Bir hatayı (bugfix) giderdiğimizizi düşünelim. Bu bugfix için oluşturduğumuz commitin hangi sürümlerde olduğunu yani hangi müşteriye gönderildiğini şu şekilde öğrenebiliriz:

```
$ git tag --contains f78j890
v1.5
v1.1
```

Alias direktifi ile sıkca kullandığımız komutlar için kısa isimler oluşturabiliriz. Aşağıda yer alan örnekte çalışma sahasındaki değişiklikleri görmek için git status yerine git st girmek yeterli olacaktır.

```
$ git config --global alias.st status
$ git st
```

Bir dal bünyesinde neler olup, bittiğini şu şekilde öğrenebiliriz:

```
$ git log --walk-reflogs master

$ git log --walk-reflogs
commit eb464a4167b7eeb175a6935542d3eee18d09e552
Reflog: HEAD@{0} (Özcan Acar <acar@agilementor.com>)
Reflog message: commit: dosyal
Author: Özcan Acar <acar@agilementor.com>
Date:   Sat Aug 22 14:20:40 2015 +0200

    dosyal eklendi

commit 9f50e743bf667b09a447aa225660aaccf08fd617
Reflog: HEAD@{1} (Özcan Acar <acar@agilementor.com>)
Reflog message: commit: 9999#
Author: Özcan Acar <acar@agilementor.com>
Date:   Sat Aug 22 14:16:59 2015 +0200

    cleanups

commit fa5c955730e21b3c703f522c10844ef0a7b7e474
Reflog: HEAD@{2} (Özcan Acar <acar@agilementor.com>)
Reflog message: commit: xxxxx
Author: Özcan Acar <acar@agilementor.com>
Date:   Sat Aug 22 14:15:49 2015 +0200

    initial commit
```

Aşağıda komut bize bu listenin kısaltılmış halini verecektir:

```
$ git reflog
eb464a4 HEAD@{0}: commit: dosyal
9f50e74 HEAD@{1}: commit: dosyal eklendi
fa5c955 HEAD@{2}: commit: initial commit
```

Git Hooks

.git/hooks dizininde belli aşamalarda tetiklenebilecek program parçaları (script) yer almaktadır. Init ile yeni inşa edilen bir depoda aşağıda yer alan programlar bulunmaktadır.

Bu programların aktif hale gelebilmesi için .sample ekinin silinmesi gerekmektedir. Örneğin commit-msg.sample dosyasını commit-msg olarak değiştirdiğimiz taktirde, yaptığımız her commit enasında bu dosyada bulunan program devreye girerek, belli işlemler yapılmasını sağlayacaktır. Örneğin bu şekilde belli kelimeler ihtiva eden commit mesajları geri çevrilebilir.

```
$ ls -l .git/hooks/
applypatch-msg.sample
commit-msg.sample
post-update.sample
pre-applypatch.sample
pre-commit.sample
pre-push.sample
pre-rebase.sample
prepare-commit-msg.sample
update.sample

$ cat git/hooks/commit-msg.sample
#!/bin/sh
#
# An example hook script to check the commit log message.
# Called by "git commit" with one argument, the name of the file
# that has the commit message. The hook should exit with non-zero
# status after issuing an appropriate message if it wants to stop the
# commit. The hook is allowed to edit the commit message file.
#
# To enable this hook, rename this file to "commit-msg".

# Uncomment the below to add a Signed-off-by line to the message.
# Doing this in a hook is a bad idea in general, but the prepare-commit-msg
# hook is more suited to it.
#
# SOB=$(git var GIT_AUTHOR_IDENT | sed -n 's/^\(.*>\).*$/Signed-off-by: \1/p')
# grep -qs "^$SOB" "$1" || echo "$SOB" >> "$1"

# This example catches duplicate Signed-off-by lines.

test "" = "$(grep '^Signed-off-by: ' "$1" |
    sort | uniq -c | sed -e '/^[ ]*1[ ]/d')" || {
    echo >&2 Duplicate Signed-off-by lines.
    exit 1
}
```


Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Git stash yapılan değişikliklerin geçici olarak rafa kaldırılmalarını mümkün kılmaktadır.
 - Commit öncesi indexe dosya ekleme işlemi interaktif yapılabilmektedir. Bu yapılan değişiklikleri değişik commit nesneleri bünyesinde gruplamak için kullanılabilecek bir yöntemdir.
 - Commit, push ya da pull gibi işlemler öncesi ya da sonrasında hook ismi verilen program parçalarını (bash script) koşturmak mümkün. Örneğin bir commit öncesinde commit mesajında bir yazım hatası olup, olmadığını kontrol ettikten sonra, commit sonrasında tüm ekibe yeni commit yapıldığını bildiren bir e-posta gönderilmesini sağlayabilirsiniz.
-

8. Bölüm

Depo Bağımlılıkları

Submodül Kullanımı

Bir Git deposunu bir sürüm birimi (release unit) olarak düşünebiliriz. Bir depo bünyesinde bir proje yer alır. Yapılan her commit, oluşturulan her etiket ya da dal sadece bu depo bünyesinde geçerlidir. Bazı şartlarda bir proje birden fazla modülden oluşabilir. Örneğin üzerinde çalıştığım güncel proje sunucu tarafındaki işletme mantığını tutan (backend) bir modül, kullanıcı arayüzünü ihtiva eden ikinci bir modülden oluşmaktadır. Proje yapısını şu şekilde düşünebiliriz:

```
Proje
|
|-- backend
|   |
|   |-- src
|   |
|   |-- test
|   |
|   |-- config
|   |
|   |-- build
|   |
|-- frontend
    |
    |
    |-- web-app
    |
    |-- test
    |
    |-- config
    |
    |-- build
```

Bu projenin aslında iki modülden oluştuğunu görmekteyiz. Lakin bu iki modülün aynı proje bünyesinde yer alması, yapılan her commit ile aynı geçmişe sahip olacakları anlamına gelmektedir. Örneğin backend için yaptığım commit aynı zamanda frontend dosyalarını da kapsayacaktır. Sadece frontend bünyesinde olup, bitenleri bilmek istemiş olsaydım, iki modülün ortak geçmişine bakmak kafa karıştırıcı olabilirdi. Bu yüzden bu iki modülün ayrıştırılarak, iki depoya yerleştirilmelerinde fayda vardır. Hadi şimdi bunu yapalım:

```
$ git init backend
Initialized empty Git repository in c:/Users/acar/temp/backend/.git/

$ cd backend
```

```
$ echo "# backend" >> README.md

$ git add README.md

$ git remote add origin https://github.com/ozcanacar/backend.git

$ git remote -v
origin https://github.com/ozcanacar/backend.git (fetch)
origin https://github.com/ozcanacar/backend.git (push)

$ git commit -m "initial commit"
[master (root-commit) 6840068] initial commit
warning: LF will be replaced by CRLF in README.md.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 README.md

$ git push origin master
Username for 'https://github.com': ozcanacar
Password for 'https://ozcanacar@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 228 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ozcanacar/backend.git
 * [new branch]      master -> master
```

Şimdi aynı işlemi frontend projemiz için yapalım:

```
$ git init frontend
Initialized empty Git repository in c:/Users/acar/temp/frontend/.git/

$ cd frontend/

$ echo "# frontend" >> README.md

$ git add README.md

$ git commit -m "initial commit"
[master (root-commit) 6e79a9b] initial commit
warning: LF will be replaced by CRLF in README.md.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 README.md

$ git remote add origin https://github.com/ozcanacar/frontend.git
```

```
$ git push origin master
Username for 'https://github.com': ozcanacar
Password for 'https://ozcanacar@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 229 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ozcanacar/frontend.git
 * [new branch]      master -> master
```

Şu şekilde bir projeyi iki depoda yer alan iki modüle ayırmış olduk. Frontend yapı itibari ile backend modülüne bağımlı. Bu yüzden frontend üzerinde çalıştığım zaman, backend projesinin de elimin altında olmasını istiyorum, çünkü bazen frontend modülünün ihtiyaç duyduğu kodu backend bünyesinde implemente etmem gerekiyor. Bu durumda her iki depoyu da klonlamam gerekmektedir. Bu mümkün, lakin Git submodule mekanizması yardımı ile iki modülü tek bir depoda bir araya getirmek mümkün. Bunun nasıl yapıldığını şimdi bir örnek üzerinde inceleyelim.

Öncelikle frontend deposunun bir klonunu oluşturuyorum:

```
$ git clone https://github.com/ozcanacar/frontend.git
Cloning into 'frontend'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

Şimdi backend projesini submodule olarak frontend deposuna ekleyelim:

```
$ cd frontend/

$ git submodule add https://github.com/ozcanacar/backend.git backend
Cloning into 'backend'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory.

$ ls -l
total 1
-rw-r--r--  1 acar      Administ    12 Aug 22 07:04 README.md
drwxr-xr-x  4 acar      Administ    0 Aug 22 07:09 backend
```

Bu işlemin ardından frontend dizininde .gitmodules isminde bir dosya yer alacaktır ve içeriği şu şekilde olacaktır:

```
$ cat .gitmodules
[submodule "backend"]
    path = backend
    url = https://github.com/ozcanacar/backend.git
```

Oluşturduğumuz bu yapıyı bir commit nesnesi ise deponun bir parçası haline getirmemiz gerekiyor.

```
$ git status -s
A .gitmodules
A backend

$ git commit -m "backend modülü eklendi"
[master a556801] backend modülü eklendi
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory.
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 backend
```

Bu commit ile .gitmodules dosyasını ve backend submodülünü depoya eklemiş olduk. Ls-tree ile master dalına göz attığımızda, backend dizininin de commit nesnesinin bir parçası olduğunu görmekteyiz.

```
$ git ls-tree master -r
100644 blob 5b77a63f06d0f26fbe5ea9591f6b19d737e0284a    .gitmodules
100644 blob 89ce48d30213468b3d4d8d3e8feaf5dd83ccf26d    README.md
160000 commit 684006874fc4184c0644b4092e78960934a7ad43    backend
```

Bu durumda backend dizininde kullandığımız her Git komutu sadece backend modülü için geçerli olacaktır. Örneğin backend projemize yeni bir dosya ekleyelim:

```
$ cd backend/

$ echo "test" > test.txt

$ git add test.txt

$ git commit -m "test.txt eklendi"
[master 5278c26] test.txt eklendi
```

```
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 test.txt
```

Şimdi bu modülün commit geçmişine bir göz atalım:

```
$ git log --oneline
5278c26 test.txt eklendi
6840068 initial commit
```

Şimdi bir üst dizine yani frontend modülüne geçiyoruz ve aynı işlemi orada da yapıyoruz:






```
$ cd ..

$ git log --oneline
a556801 backend modülü eklendi
6e79a9b initial commit
```

Görüldüğü gibi iki modülün commit geçmişi farklı. Oluşturduğumuz bu submodül yapısını push ile Github bünyesindeki frontend deposuna aktarabiliriz:

```
$ git push origin master
Username for 'https://github.com': ozcanacar
Password for 'https://ozcanacar@github.com':
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 402 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ozcanacar/frontend.git
6e79a9b..a556801 master -> master
```

Şimdi Github.com kullanıcı arayüzünden frontend deposunun içeriğine bir göz atalım:

backend modülü eklendi		
 ozcanacar authored 15 minutes ago		latest commit a556801624 
 backend @ 6840068	backend modülü eklendi	15 minutes ago
 .gitmodules	backend modülü eklendi	15 minutes ago
 README.md	initial commit	an hour ago

Resim 1

Resim 1 de görüldüğü gibi frontend deposu bünyesinde backend deposunun 6840068 hash değerindeki commit nesnesine referans bulunmaktadır. Backend deposuna göz attığımızda, bunun bizim oluşturduğumuz ilk commit nesnesinin hash değeri olduğunu görmekteyiz:

```
$ git log --oneline
6840068 initial commit
```

Git, submodüllerden oluşan bir depoda modüller arasındaki ilişkiyi commit nesneleri üzerinden sağlamaktadır. Bu şekilde örneğin frontend projesinin belli bir backend versiyonu ile ilişkilendirmek mümkündür. Bunu hem commit bazında hem de etiket bazında yapabiliriz. Aşağıdaki örnekte frontend modülünü backend modülünün v1.5 sürümü ile ilişkilendiriyoruz.

Öncelikle backend modülünde v1.5 isimli etikete geçiş yapıyoruz:

```
$ git checkout v1.5
Note: checking out 'v1.5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at 500e859... test
```

Şimdi frontend modülüne geçelim ve ne durumda olduğumuza bir göz atalım:

```
$ cd ..

$ git status
On branch master
Your branch is up-to-date with 'origin/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:   backend (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Frontend modülündeyken status komutu ile son durumumuza göz attığımızda, backend dizininin değişikliğe uğradığını görmekteyiz. Bunun sebebi, bizim bu modül bünyesinde v1.5 etiketine geçmiş olmamızdır. Burada v1.5 etiketi bir commit nesnesine işaret etmektedir ve checkout komutu ile aslında dalın bu commit nesnesine geri dönmüş olduk. Bu sebepten dolayı zaten detached HEAD durumuna düştük.

Bu durumda frontend modülünü ya da deposunu backend modülü ya da deposunun v1.5 sürümü ile ilişkilendirmek için aşağıdaki işlemleri gerçekleştiriyoruz:

```
$ git add backend

$ git commit -m "backend v1.5"
[master 50dc07a] backend v1.5
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push
Username for 'https://github.com': ozcanacar
Password for 'https://ozcanacar@github.com':
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 252 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/ozcanacar/frontend.git
 0ea0c2b..50dc07a  master -> master
```

Bu şekilde frontend deposunu backend deposunun v1.5 etiketindeki hali ile ilişkilendirmiş olduk. Şimdi submodüllerden oluşan bir depoyu nasıl klonlayarak, kullanabileceğimize bir göz atalım.

Öncelikle frontend deposunu klonluyoruz:

```
$ git clone https://github.com/ozcanacar/frontend.git frontend-backend
Cloning into 'frontend-backend'...
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 17 (delta 5), reused 13 (delta 1), pack-reused 0
Unpacking objects: 100% (17/17), done.
Checking connectivity... done.

$ cd frontend-backend/

$ ls -l
total 1
```

```
-rw-r--r--    1 acar    Administ    12 Aug 22 07:57 README.md
drwxr-xr-x    2 acar    Administ      0 Aug 22 07:57 backend
-rw-r--r--    1 acar    Administ    12 Aug 22 07:57 test.txt

$ cd backend/

$ ls -l
total 0
```

Klonladığımız deponun backend dizinine bir göz attığımızda, bu dizinin boş olduğunu görmekteyiz. Submodüllerden oluşan depolarda Git klonlama işlemi esnasında submodülleri klonlamaz. Şu şekilde submodülleri klonlamamız gerekmektedir:

```
$ git submodule update --init
Submodule 'backend' (ozcanacar/backend.git) registered for path 'backend'
Cloning into 'backend'...
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 0), reused 12 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), done.
Checking connectivity... done.
Submodule path 'backend': checked out '500e859f2a766510374cf916b8a4f4afb7b1b7de'
```

Backend modülünün 500e859 commit nesnesine işaret ettiğini görmekteyiz. Bu v1.5 etiketinin işaret ettiği commit nesnesidir:

```
$ git log --oneline
500e859 v1.5 etiketi oluşturuldu
5278c26 test.txt eklendi
6840068 initial commit

$ git show-ref --tags
500e859f2a766510374cf916b8a4f4afb7b1b7de refs/tags/v1.5
```

Görüldüğü gibi frontend ile backend arasındaki ilişki, daha önce bu iki modül arasında tayin ettiğimiz v1.5 etiketi ile eş değerdedir. Init öncesinde hangi submodüllerin kullanımda olduğuna şu şekilde bir göz atabiliriz:

```
$ git submodule status
-500e859f2a766510374cf916b8a4f4afb7b1b7de backend
```

Eksi işareti backend modülünün henüz init ile alınmadığını göstermektedir. Init işleminden sonra liste şu şekilde olacaktır:

```
$ git submodule status
500e859f2a766510374cf916b8a4f4afb7b1b7de backend (v1.5)
```

Backend dizini bünyesinde istediğimiz türde değişiklik yapabilir, istediğimiz dalları kullanabiliriz. Bu değişiklikler frontend deposunu doğrudan etkilemez. Lakin örneğin backend deposunda başka bir dala geçmemiz frontend deposunda bulunan kodları etkileyebilir ya da çalışmaz hale getirebilir. Bu durumda submodule update komutu yardımıyla iki depo arasındaki ilişkiyi tekrar olması gerektiği seviyeye şu şekilde getirebiliriz:

```
$ git submodule update
Submodule path 'backend': checked out '500e859f2a766510374cf916b8a4f4afb7b1b7de'
```

Git bu örnekte backend deposunda şu komutu kosturarak, depoyu v1.5 etiketindeki commit seviyesine getirmiştir:

```
$ git checkout 500e859f
```

Frontend deposundan backend submodülünü silmek için aşağıdaki işlemleri gerçekleştirmemiz gerekiyor:

- .gitmodules dosyasından submodül bölümünün silinmesi
- .git/config dosyasından submodül ile ilgili bölümlerin silinmesi
- git rm -cached backend ile submodülün fiziksel olarak silinmesi
- yaptığımız değişikliklerin bir commit ile depoya aktarılması.

Submodüller aracılığı ile depoları ilişkilendirmek kolay gibi görünse de, bazı zorlukları beraberinde getirdiğini de belirtmemiz de fayda var. En büyük sorunu klonlama işlemi esnasında submodüllerin de otomatik olarak klonlanmaması teşkil ediyor. İkinci bir komut yardımı ile submodül içeriğini temin etmemiz gerekiyor. Bunun yanı sıra submodüllerden oluşmuş bir projenin tüm kodunu GitHub'dan edinmemiz mümkün değil, çünkü depolar arasındaki ilişki sadece commit referansları aracılığı ile oluşturuluyor. Dal işlemlerinin de submodüller bünyesinde senkron işlemesi gerekiyor, çünkü burada da Git dallar arası otomatik geçiş sağlamıyor. Git subtree mekanizması burada bir alternatif olabilir. Bu mekanizmanın nasıl işlediğini şimdi yakından inceleyelim.

Subtree Kullanımı

Bir projenin birden fazla depoya dağıldığını düşünelim. Örneğin bir önceki bölümde gördüğümüz gibi projemiz frontend ve backend modüllerinden oluşuyor olsun. Bu iki modülü birbirinden

bağımsız iki Git deposunda tutabiliriz. Subtree mekanizmasını kullanarak, bu iki modülü bir depo bünyesinde tek bir proje geçmişine sahip olacak şekilde bir araya getirmek de mümkün. Şimdi bunun nasıl yapıldığını bir örnek üzerinde inceleyelim.

Öncelikle frontend deposunu klonluyoruz:

```
$ git clone https://github.com/ozcanacar/frontend.git
Cloning into 'frontend'...
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 17 (delta 5), reused 13 (delta 1), pack-reused 0
Unpacking objects: 100% (17/17), done.
Checking connectivity... done.
```

Şimdi bu depoya backend deposunda yer alan kodları entegre edelim:

```
$ git subtree add --prefix=backend https://github.com/ozcanacar/backend.git master
git fetch https://github.com/ozcanacar/backend.git master
warning: no common commits
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 15 (delta 0), reused 15 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), done.
From https://github.com/ozcanacar/backend
 * branch          master      -> FETCH_HEAD
Added dir 'backend'
```

Deponun commit geçmişi şimdi şu şekilde olacaktır:

```
$ git log --oneline
d5d2a91 Add 'backend/' from commit 'ae72a0b38fcbe8add19f4af42a7c7ecce172a090'
cb650e5 cleanups
ae72a0b test
50dc07a backend v1.5
```

Backend deposu frontend deposuna backend dizini aracılığı ile birebir entegre edildi. İki deponun entegrasyonu esnasında backend deposunda bulunan tüm commitler frontend deposunun geçmişine eklendi ve birleşmeyi sembolize eden d5d2a91 commit nesnesi oluşturuldu. Bu noktadan itibaren deponun yeni yapısını sadece bir depo üzerinde işlem yapıyormuş gibi düşünebiliriz. Frontend üzerinde yaptığımız her commit ve push, backend dizinindeki tüm dosyalar ve değişikliklerle birlikte frontend deposuna eklenecektir. Backend dizininde herhangi bir dosyayı değiştiresek bile, oluşturacağımız commit nesnesi frontend deposunun geçmişine eklenecektir.

Zaman içinde backend deposunda meydana gelen değişiklikleri (başka bir ekip tarafından yapılmış olabilirler) tekrar frontend deposuna şu şekilde entegre edebiliriz:

```
$ git subtree pull --prefix=backend https://github.com/ozcanacar/backend.git master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ozcanacar/backend
 * branch                master      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 backend/xx | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 backend/xx

$ git log --oneline
8275199 Merge commit '39bdfae988a059ee0183b7be926c7b20cbc2a'
39bdfae backend REST API eklendi
d5d2a91 Add 'backend/' from commit 'ae72a0b38fcbe8add19f4af42a7c7ecce172a090'
cb650e5 cleanups
ae72a0b test
50dc07a backend v1.5
```

Subtree pull komutuyla backend deposundaki yeni commitler frontend deposuna aktarılmıştır. Frontend deposunun geçmişinde bu işlemlerden doğan neticeleri görmekteyiz. Peki backend bünyesinde yaptığımız değişiklikleri frontend değil de, backend deposuna nasıl aktarabiliriz? Şu şekilde:

```
$ cd backend

$ touch dosyal.txt

$ git add dosyal.txt

$ git commit -m "backend dosyal.txt eklendi"

$ git subtree push --prefix=backend https://github.com/ozcanacar/backend.git master
git push using: https://github.com/ozcanacar/backend.git master
1/      8 (0)2/      8 (0)3/      8 (0)4/      8 (1)5/      8 (2)6/      8 (3)7/
8 (4)8/      8 (5)Username f
Password for 'https://ozcanacar@github.com':
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 300 bytes | 0 bytes/s, done.
```

```
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/ozcanacar/backend.git
   39bdfae..cdbc21e  cdbc21e941a7a481ce363edfb20544045c518cc5 -> master
```

Burada subtree push komutu backend dizininde yaptığımız değişiklikleri doğrudan backend deposuna aktarmaktadır. Eğer backend dizininde hiçbir değişiklik yapmamış, buna karşın frontend dizininde değişiklikler yapmış olsaydık, bu değişiklikler backend deposunu ilgilendirmediği için subtree push ile backend deposuna aktarılacak bir değişiklik olmazdı.

En son örnekte git subtree push sonrasında git push yaptığımız takdirde backend dizininde yaptığımız değişiklikler frontend deposuna da aktırılır.

Submodule ve subtree arasındaki farklılıkları şu şekilde özetleyebiliriz:

- Submodule fiziksel olarak depoları birleştirmez, sadece commit nesne referansları aracılığı ile ilişkilendirir. Buna karşın subtree bir depoya başka bir deponun içeriğini import eder.
- Submodule kullanımında iki deponun birbirlerinden bağımsız commit geçmişi vardır. Subtree kullanımda tek bir depo ve commit geçmişi oluşur.

Özet

Bu bölümde incelediğimiz konuları şu şekilde özetleyebiliriz:

- Birbirlerine bağımlılığı olan depoları Git submodül tekniğini kullanarak, bir depo bünyesinde bir araya getirmek mümkündür. Bu teknik bir araya getirilen depolardan fiziksel yeni bir depo oluşturmaz. Daha ziyade submodüller birbirlerinden bağımsızdırlar ve kendi depo geçmişlerini korurlar.
- Subtree yönteminde mevcut bir depo başka bir depoya fiziksel olarak dahil (import) edilir. Bu iki deponun aynı depo geçmişine sahip olacağı anlamına gelmektedir.

BTSoru.com

BTSoru.com yazılımcıları bir araya getirmek için kurduğum bir soru-cevap platformu. Bu kitap hakkındaki sorularınızı BTSoru.com üzerinden [bana](#) yöneltebilirsiniz.

The screenshot shows the BTSoru.com website interface. At the top, there's a navigation bar with links for 'sorular', 'etiketler', 'kullanıcılar', 'madalyalar', and 'cevapsız sorular'. A 'Yeni bir soru sor' button is also present. Below the navigation bar, there's a search bar with a 'arama' button and radio buttons for 'sorular', 'etiketler', and 'kullanıcılar'. The main content area displays a list of questions under the heading 'Tüm Sorular'. Each question entry includes the number of votes, answers, and views, the question text, tags, and the user who asked it. On the right side, there's a sidebar with a 'Türkiye Yazılımcı Raporu 2012' section, a 'BTSoru.com ?' section showing '2076 soru' and '3445 cevap', and an 'En yeni etiketler' section listing various tags like 'unauthorized', 'geonetwork', 'www', 'jasperserver', etc.

BTSoru.com

Tüm Sorular

0 oy 1 cevap 7 gösterim **javax.ejb.EJBException: javax.ejb.EJBException: javax.ejb.CreateException: Could not create stateless EJB**
6 dakika önce 74n3r 1

0 oy 0 cevap 56 gösterim **Hibernate ve Spring üzerinde çalışan bir proje için dil ayarları nasıl olmalıdır ?**
1 saat önce aheng 191

0 oy 0 cevap 5 gösterim **Yazılım güvenliğinde ESAPI JAVA kullanımı**
1 saat önce hale 127

0 oy 0 cevap 9 gösterim **Geonetwork Metadata insert işlemi unauthorized hatası**
17 saat önce jacksparrow47 156

0 oy 0 cevap 13 gösterim **timeout expired hatası**
19 saat önce ibat90 1

0 oy 1 cevap 54 gösterim **Orta ölçekli bir Android projesi önerir misiniz?**
22 saat önce juanov 80

0 oy 2 cevap 53 gösterim **Web projemdeki hataları düzeltmek**
22 saat önce macroasm 6

0 oy 2 cevap 64 gösterim **Android'te aynı anda birden fazla animasyonu nasıl çalıştırabilirim ?**
dün sgurdag 1

0 oy 7 cevap 1.3k gösterim **Xades ile xml imzla işlemi nasıl yapılıyor?**
dün otaskiran 1

Türkiye Yazılımcı Raporu 2012

BTSoru.com ?

2076 soru
3445 cevap
en son güncellenen sorular

En yeni etiketler

unauthorized geonetwork www jasperserver animation activity parsing ayırtma facebook_user_id konu adobe fireworks cas mobil-uygulama mobil-yazılım jpqi volatileimage bytea aracı sqlmanager lightbox routers procedure error easyui gizliservis popüler etiketler

KurumsalJava.com

KurumsalJava.com adresinde blog yazılarım yer almaktadır. Java ve Spring konularındaki yazılarımı buradan takip edebilirsiniz.

KurumsalJava.com'da yer alan yazılarımı bir kitap haline getirdim. PDF formatındaki bu kitabı [buradan](#) edinebilirsiniz.

Kurumsal Java

Java Enterprise Architecture

[RSS](#)
[Yorum](#)
[1395](#)
[Followers](#)

[ANA SAYFA](#)
[DANIŞMANLIK](#)
[HAKKIMDA](#)
[İÇERİK](#)
[İLETİŞİM](#)
[KOD KATA](#)
[MEDYA](#)
[YAZILIM METOTLARI](#)
[YAZILIMCI RAPORU](#)

[Haberler](#)
[Son Yazılar](#)



KurumsalJava.com Kitapı

KurumsalJava.com bünyesinde yazdığım yazılardan seçtiğim elli yazıyı bu e-kitapta bir araya getirdim. Beğeninize sunarım.

[f](#)
[t](#)
[ff](#)
[in](#)
[g](#)
[s](#)

Bilişim Soru & Cevap Platformu

BTSORU?

E-POSTA BİLGİLENDİRME

Yeni yazılardan haberdar olmak için lütfen e-posta adresinizi giriniz.

You may manage your subscription options from your [profile](#).

RASTGELE ALINTI

"Tek bir dili savunan yazılımcılar uzman, çok dili kullananlar ustadır. Tercih sonradan gelen olmalı."

— Özcan Acar, <http://goo.gl/NLyyu8>

ONLINE ÜYELER

[3 kullanıcı Online](#)

Özcan Acar, 2 ziyaretçi

SON YAZILAR

HABERLER

KurumsalJava.com Kitapı

KurumsalJava.com bünyesinde yazdığım yazılardan seçtiğim elli yazıyı bu e-kitapta bir araya getirdim. Beğeninize sunarım. [download id="75"]

Spring Core Sertifika Sınavı Ardından

Geçen sene katıldığım Spring Integration ve Spring Core kurslarının ardından bu senein mayıs ayında [Spring Integration sertifikasını almıştım](#). Katıldığım kurslardan sonra aklımda Pratik Spring Core kitabını yazma fikri oluştu. Kitabı tamamladım ve yakında pragmatikprogramci.com adresi

EOF (End Of Fun)

Yolculuğumuzun sonuna geldik. Sizin için burası son durak değil. Git hakkında internette birçok faydalı kaynak bulabilirsiniz. Aklınıza takılan soruları BTSoru.com üzerinden benimle ve diğer bilişimci arkadaşlarla paylaşabilirsiniz.

Sağlıkcakla kalın. Her şey gönlünüzce olsun.

EOF (End Of Fun)

Özcan Acar

<http://www.kurumsaljava.com>

<http://www.ozcanacar.com>

<http://www.twitter.com/oezcanacar>

<https://www.facebook.com/oezcanacar>

<http://www.linkedin.com/pub/%C3%B6zcan-acar/50/550/2a3>