GIT Püf Noktaları

Uğur “vigo” Özyılmazel

İçindekiler

GIT Püf Noktaları

Revizyon kontrol sistemi olarak son yıllara damgasını vurmuş bir araç GIT. Bu kitapta gündelik geliştirme rutinlerinde sıkça karşınıza çıkabilecek konulara değineceğim.

Ek olarak, zaman içinde topladığım, notlarımdan derlediğim püf noktalarını da paylaşacağım.

Bölüm 01

Bölüm 02

- Branch'leri Birleştirmek
- Branch'lerin Çakışması: Conflict
- Branch'leri Birleştirmek: `rebasing`
- Branch Rebase Sırasında Çakışma: **Rebase Conflict**
- Değişiklikleri Görüntülemek: `git diff`
- Etiketlemek Nedir?: `git tag`

Bölüm 03

- Commit'leri Birleştirmek: **Interactive Rebasing**
- Commit'leri Bölmek
- Cımbızla Commit'i Almak: **Cherry Picking**
- Hataları İşlemleri Geri Almak ya da Vazgeçmek: `reset revert amend`
- Commit'e Not Eklemek
- Her şey Kayıt Altında! En az 90 gün: `git reflog`

Bölüm 04

- Remote Kavramı Nedir? Remote'larla Çalışmak
- Kendi GIT Reponuzu Yapın!
- GitHub, BitBucket ve GitLab ile Çalışmak
- Repo içinde Repo: `git submodule`

Bölüm 05

- Commit Öncesi ya da Sonrası Otomasyonu: **GIT Hook'ları**
- Bundle Nedir?
- Commit'inizi İmzalayın
- Revizyonları Sorgulamak
- Commit'leri Sorgulamak: `blame`
- Bisect Nedir?
- Yardımcı Araçlar
- Faydalı İpuçları

Hatırlatmalar

- Neticede ben yazar değilim. Yüksek ihtimal çok sayıda imla ve yazım hatası yapacağım. Bu kitap açık-kaynak olarak GitHub'da bulunuyor. Yardım edip hataları düzeltmeme yardımcı olursanız süper olur.
- Biliyorum çok kızan olacak ama bu kitapta pek çok yerde yarı İngilizce yarı Türkçe kelimeler olacak. Çevirebildiklerimi çevireceğim. Bazı durumlarda çevirmek ve doğru anlamı bulmak zor oluyor. İdare edin :)
- Kitabı online olarak okumak için tıklayın.

Teknik Eleştiri / İnceleme

Açıklama

Bu kısım, okuduğum yabancı kitapların (*teknik konular, yazılım geliştirme ve benzeri*) çoğunda gördüğüm ama ne yazıkki bizim yayın-evleri ya da yazarlarımızın pek de yapmadığı kısım. **Techinal Review** yani ilgili konunun uzmanı ya da sektör profesyoneli tarafından yapılan yorum.

Bunun çok önemli olduğunu düşünüyorum. Neden mi?

- Acaba gerçekten konuyu doğru anlamış ve anlatmışmıyım?
- Yazdıklarım arasında tutarsızlıklar var mı?
- Verdiğim örnekler doğru mu? Ya da anlattığım konu ile ilgili mi?
- Hangi kısımları teknik anlamda düzeltmek gerek?

sonuç olarak bu kitap gerçekten amacına hizmet etsin istiyorum.

İşte bu doğrultuda bize yardımcı olacak kişi sevgili Lemi Orhan Ergin.

İnceleme ve Yorum

@wip (Lemi)

Bölüm 01

Bu bölümde işleyeceğimiz konular:

GIT Nedir?

Eğer komut satırından `man git` derseniz, karşınıza çıkacak olan man page'de:

```
NAME
   git - the stupid content tracker
```

ifadesini görürsünüz. **the stupid content tracker** yani: aptal içerik takipçisi. Biraz ilginç değil mi? Yazılım dünyasında Microsoft'undan Apple'ına Google'ına kadar neredeyse 7'den 70'e kullandığımız bir araç var ve adı: aptal içerik takipçisi...

GIT aslında, dağıtık çalışan sürüm kontrol sistemi (*DVCS*) ve kaynak kod yönetim (*SCM*) aracıdır. DVCS: **D**istributed **V**ersion **C**ontrol **S**ystem, SCM: **S**ource **C**ode **M**anagement anlamına gelir. Eşdeğer diğer araçlardan öne çıkan farkları ise;

- Herhangi bir merkez sunucuya ihtiyaç duymadan, offline olarak çalışabilmesi
- Güvenilirlik, **commit**'lerin tekil olması [1]
- Hızlı olması [2]
- Az yer tutması
- Sıfır maliyetle **branching** (*dallanma*) yapabilmek ve **merge** etmek (*birleştirebilmek*)
- **Deployment** ve benzeri işler için de kullanılması.

Bu güzel tool, Linux'un çekirdeğini yazan Linus Torvalds tarafından geliştirilmiş ve açık-kaynak şeklinde dağıtılmıştır. Tüm kaynak kod GitHub'da durmaktadır. Bu kitabı yazdığım an itibariyle aktüel olan versiyon: **2.13.1**

> [1]. Mart 2017 itibariyle biraz can sıkıcı bir durumla karşılaştı GIT kullanıcıları. ↩

> [2]. Facebook / Google ve benzeri ölçeklerde projelerde (*gigabyte'larca*) bazı işlemler çok yavaşlıyormuş. ↩

GIT'in Hikayesi

Linus Torvalds, 2002 yılında, kernel'i geliştirirken BitKeeper adlı revizyon kontrol sistemini kullanıyor. 2005 yılında Linux kernel geliştirici topluluğu ile BitKeeper arasında bir sıkıntı oluyor. BitKeeper'ın ücretsiz kullanım lisansı iptal oluyor ve Linus'un sigorta bu noktada atıyor.

İzlediğim Git Under the Hood eğitiminde Jeffrey Haemer, Linus'un GIT'i geliştirmeye bir pazartesi başlayıp, çarşamba gününden itibaren GIT'i GIT ile versiyonlamaya başladığını söyledi. Bu kadar hızlı geliştirmesinin sebebi olarak da zaten çok iyi bildiği kernel'i kopyalaması olduğunu söyledi.

Hatta Ken Thompson'ın da Unix'i **tam 1 ay**da yazdığını, Linus'un da Unix'i örnek aldığını söyledi...

Sonuç Olarak...

Neticede GIT aslında bir **veritabanı**dır. Tanımlandığı dizin altında çalışan, ilgili bilgilerini, ayarlarını ve benzeri bilgilerini `.git/` dizini altında tutan, o dizindeki tüm dosyaların (*eğer izole edilmemişse*) versiyonlarını yani dosyalardaki değişikliklerin tarihçesini, bu kendi şahsına özel veritabanı mekanizması içinde saklar.

Repository ya da Repo Nedir?

Revizyon kontrolü altındaki dizin bir Repository'dir. Sözlük anlamına baktığınızda; *depo*, *dolap*, *kutu*, *kab* gibi karşılığı olduğunu görürsünüz. Bence tam anlamıyla içinde dosyaların (*daha doğrusu kaynak kodların*) bulunduğu bir depodur Repository.

Halk arasında kısaca **Repo** olarak kullanılır. Film sektöründe çalışan biri için Repo tatil anlamına geldiği gibi finans / bankacılık sektöründe de *gecelik repo* gibi faiz içerikli anlamları da bulunur.

Eğer bilgisayarınızda GIT kuruluysa;

```
$ git --version
$ git help
```

gibi komutların çalıştığını göreceksiniz. Eğer yukarıdaki komutlar çalışmıyorsa, kullandığınız işletim sistemine göre; bu link yardımıyla GIT'i indirebilir ve kurabilirsiniz.

GIT, içinde harika bir dokümantasyon ile geliyor. Kullanacağınız komut hakkında bilgi almak çok kolay:

```
$ git help commit
$ git help status
$ git help clone

# git help KOMUT
```

Hızlıca repository oluşturmak için iki yöntem var; ilk yöntem, bir dizin oluşturup, içine girip repoyu oluşturmak: `git init`

```
$ mkdir proje
$ cd proje/
$ git init
Initialized empty Git repository in /private/tmp/proje/.git/
```

GIT bize **boş bir repository** oluşturduğunu söyler. Diğer yöntem ise tek satırda;

```
$ git init proje
Initialized empty Git repository in /private/tmp/proje/.git/
```

işi halletmektir. Örnekleri kendi bilgisayarımda `/tmp/` yani temporary / geçici dizinde yapmaktayım. Bu bakımdan gördüğünüz `/private/tmp/proje/` gibi dizinler sizi şaşırtmasın!

Tamam, repo oluştu. Peki sonra ? Bence en çok kullanılan komutlardan biri olan `git status` ile tanışalım.

```
$ git status

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

`git status` ile o anki durumu görüntülüyoruz. An itibariyle `master` branch'deyiz, commit edecek hiçbir şey yok. Bu durumda karşımızda iki tane yeni kavram var. `branch` ve `commit`.

Branch Nedir?

İstediğiniz bir anda, elinizdeki kod'dan hızlıca bir ya da N tane kopya çıkartma işlemidir. Yerel operasyondur. Yani yaptığınız her branch, siz paylaşmadıkça sadece sizde bulunur.

Daha teknik bir anlatımla branch aslında içinde commit-id yazan bir işaretçiden başka bir şey değildir.

GIT, sıfır bir repository oluşturulduğunda, aksi belirtilmedikçe, varsayılan (*default*) branch olarak **master** branch'i oluşturur:

Repository ya da Repo nedir? bölümündeki örneği hatırlayalım:

```
$ git init proje
Initialized empty Git repository in /private/tmp/proje/.git/

$ git status

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

şu an **master** branch'deyiz ve commit edecek hiçbir şey yok...

Konfigürasyon nedir?

3 kapsamlı konfigürasyon bulunur:

1. **Local**: Sadece içinde bulunduğunuz (*dizin*) repository için geçerlidir. Kişisel projelerinizde **kişisel** e-posta adresi kullanırken, şirket projelerinde şirket tarafından verilen e-posta adresini kullanmanız gerekebilir.
2. **Global**: Bilgisayara giriş yapmış (*login olmuş*) kullanıcının erişim yetkisi olan tüm repository'ler için geçerlidir. Değer atarken `--global` anahtar kelimesi kullanılır.
3. **System**: Bilgisayardaki tüm kullanıcıları etkileyen en üst seviyedeki konfigürasyondur. Değer atarken `--system` anahtar kelimesi kullanılır.

Konfigürasyon ayarlarını yapmak için; `git config` komutunu kullanırız. İlk GIT kurulumunda olmazsa olmaz olan iki tane konfigürasyon öğesi bulunur:

1. `user.name`
2. `user.email`

Konfigürasyon İşlemleri

Sonuç olarak konfigürasyon dosyası adı üzerinde bir dosya. Herhangi bir text editör ile açıp düzenleyebilirsiniz. En hızlı ve kolay yolu bu. Bunun dışında, `git config` komutuna çeşitli parametreler geçerek değerleri sorgulayabilir, silebilir, düzeltebilirsiniz. Çok daha fazla detay için tıklayın.

Comment Out (*Yorum*)

Bu dosyada **comment out** yani programlama dillerindeki gibi yorum satırı ya da bazen bir şeyleri denemek için anlık satırı **off** etmek için popüler programlama dillerinden alışkın olduğumuz `#` ve `;` kullanılmış:

```
# bu
# yorum
# satırı
; bu
; satır da yorum...
[user]
    name = Uğur Özyılmazel
```

Değer Okumak: `--get`

Örneğin **core** grubu altında bulunan **filemode** değişkeninin değerini;

```
git config --get core.filemode
# true
```

`--get GRUP.DEĞİŞKEN` şeklinde okuruz. Aslında `--get` opsiyonel. Yani yazmak zorunda değilsiniz:

```
git config user.name
# Uğur Özyılmazel
```

Ben Homebrew'la beraber `bash-completion` paketini de kullandığım için, pek çok GIT işlemini **auto-complete** ya da **tab-completion** ya da otomatik olarak `TAB` tuşuna basınca tamamlama ile kullanıyorum. İşlerim süper kolaylaşıyor ve daha az şey ezberlemek durumunda kalıyorum.

Linux türevleri genelde GIT paketini kurunca otomatik olarak bu tamamlamayı da beraberinde getiriyor.

Değeri Silmek: `--unset`

Şimdi denemek için kullanıcı seviyesinde değer ataması yapalım:

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

bu komut ile kısa yol tanımladık. artık `git status` yerine `git s` yapmak yeterli. Bu **alias** konusuna ileride daha detaylı gireceğiz. Şimdi bu kısa yolu silmek için:

```
git config --global --unset alias.s
```

yapmak yeterli.

`include` ve `includeIf`

Yanılmıyorsam GIT versiyon 2.10+ ile gelen, benim çok sevdiğim hayatımı kolaylaştıran 2 direktif. Benim gibi evde kişisel bilgisayar, işte ofis bilgisayarı kullanıyorsanız aslında 2 farklı insan gibisiniz.

Belki birden fazla bilgisayarınız var. Duruma göre her bilgisayarın özel bir konfigürasyon direktifine ihtiyacı olabilir.

Ya da sadece belli dizinlerin altında belli ayarlar çalışsa?

Her bilgisayarda ortak `~/.gitconfig` dosyasını kullanmak ama bilgisayarına göre ayar yapmak istiyordum. İşte bu durumda imdadıma `include` yetişti.

```
[include]
    path = ~/.gitlocalconfig
```

`~/.gitlocalconfig` bu dosya her iki bilgisayarımda da bulunuyor. Ana GIT konfigürasyon dosyamda `user` grubunda `name` ve `email` ayarları var. Farklı bilgisayarlarda farklı gpg anahtarları kullandığım için (*bunun ne olduğunu ileride anlatacağım*) `user.signingkey` değeri her makine için farklı :)

Eğer bulunduğum dizin `~/Dev/Amiga/Bronx-Sources/` ise `~/.gitconfig-bronx-repos` konfigürasyon dosyasını kullan demek istiyorum. Nasıl mı?

```
[includeIf "gitdir:~/Dev/Amiga/Bronx-Sources/"]
    path = ~/.gitconfig-bronx-repos
```

Bu dosyada yani `~/.gitconfig-bronx-repos` dosyasında ne mi var?

```
[user]
    name = vigo/bronx
    email = vigo@bronx....org
    signingkey = D3M(O)(SC/\/3)
```

yani ben ne zaman `cd ~/Dev/Amiga/Bronx-Sources/` yapıp, bu dizin altında bulunan herhangi bir repoda işlem yapacak olursam, yukarıdaki değerler aktif olacak!

Bana şu soruyu sorabilirsiniz:

> Neden Local olarak ayar yapmadın?

İlgili dizin altın **300 tane repo** olsa? tek-tek dizinlerin altına gidip `git config` yapmak zorunda kalmak iyi bir fikir mi?

Son olarak `path = dosya` mantığında birden fazla path tanımlamak mümkün:

```
[include]
    path = ~/.git-localconfig-1
    path = ~/.git-localconfig-2
    path = ~/.git-localconfig-3
[includeIf "gitdir:~/Dev/C64/Zombie-Boys-Sources/"]
    path = ~/.git-zb-config-1
    path = ~/.git-zb-config-2
```

gibi...

Konfigürasyon Dosyası

Windows işletim sistemi dünyasındaki `.ini` formatını andıran bir deklarasyon sistemi bulunur. Dosya içinde **whitespaces** yani boşluk / alfabe dışındaki karakterler görmezden gelinir. Kabaca;

```
[bölüm]
    değişken = değer
    değişken = değer
    :
    :
```

stilindedir.

`.git/config`

Bu dosya Local yani bulunduğumuz repo altındaki `.git/` altında bulunur ve sadece o repo ile ilgili ayarları tutar. Bu kapsamda ayar yapmak için;

```
git config user.name "Bu repo için kullanacağınız AD SOYAD bilginiz"
git config user.email "Bu repo için kullanacağınız E-POSTA bilginiz"

# ya da:
git config --local user.name "Bu repo için kullanacağınız AD SOYAD bilginiz"
git config --local user.email "Bu repo için kullanacağınız E-POSTA bilginiz"

# örnek:
# git config user.name "Uğur Özyılmazel"
# git config user.email "vigo@example-local.com"
```

şeklinde işlem yapılır. `--local` anahtar kelimesi opsiyoneldir. Kullanmazsanız sıkıntı olmaz.

`~/.gitconfig`

Global dediğimiz, işletim sistemine **login** olmuş kullanıcı ile ilgili ayarların tutulduğu dosyadır. Bu kapsamda ayar yapmak için;

```
git config --global user.name "AD SOYAD"
git config --global user.email "E-POSTA"

# git config --global user.name "Uğur Özyılmazel"
# git config --global user.name "ugurozyilmazel@...com"
```

şeklinde işlem yapılır. İşin sırrı `--global` anahtar kelimesindedir.

`/etc/gitconfig`

Bu da tüm işletim sistemini etkileyen **system-wide** ayarların saklandığı dosyadır. Bu kapsamda ayar yapmak için; **sudo** yetkisi gerekir ve;

```
git config --system alias.st status
git config --system color.ui true
```

gibi işlem yapılır ve tüm kullanıcıların ortak kullanabilecekleri şeyleri ayarlamak mantıklıdır. Eğer benim gibi macOS kullanıyorsanız ve GIT'i Homebrew'dan kurduysanız, bu dosyanın bulunduğu yer: `/usr/local/etc/gitconfig` 'dir.

Temel Konfigürasyon Öğeleri

`core.editor`

`git commit` , `git merge` gibi durumlarda mesaj yazacağınız editörü ayarlamak için kullanılır. Default olarak GIT, environment'ınızdaki `$VISUAL` ya da `$EDITOR` değişkenlerine bakar.

Örneğin, editor olarak **emacs** kullanacaksanız ve;

```
export EDITOR="emacs"
```

şeklinde bir atama yaptıysanız editörünüz **emacs** olmuş demektir. Environment bazlı `$EDITOR` değişkeni örnekleri:

```
export $EDITOR="mate -w"        # TextMate
export $EDITOR="emacs"          # Emacs
export $EDITOR="vim"            # Vim
export $EDITOR="atom --wait"    # Atom
export $EDITOR="subl -n -w"     # Sublime Text
```

Eğer environment'dan ayar yapmadıysanız, GIT üzerinden de bu ayarlamayı yapabilirsiniz. Aşağıda çeşitli editör ayar örnekleri var. Hangi editörü kullanacaksanız ilgili satırı uygulayabilirsiniz.

```
git config --global core.editor emacs           # emacs
git config --global core.editor vim             # vim
git config --global core.editor "mate -w"       # TextMate
git config --global core.editor "atom --wait"   # Atom
git config --global core.editor "subl -n -w"    # Sublime
```

`color`

Renk ile ilgili işler için kullanırız. `git status` , `git diff` gibi durumlarda renkli görmek algılamamızı kolaylaştırır. Bunu aktive etmek için:

```
git config --global color.ui true
```

yapmamız gerekir. Renklerini düzenleyebileceğimiz alt başlıklardan bazıları: branch, diff, status aşağıdaki gibi ayarlanabilir.

```
git config --global color.status auto
git config --global color.diff auto
git config --global color.branch auto
```

Bu, GIT'in sizin ortamınıza göre, kullandığınız Terminal'e göre renk ayarlaması yapması anlamındadır. İsteğe göre renk değerleri vermek mümkün. Hemen `~/.gitconfig` dosyanızı açıp:

```
[color "branch"]
    current = yellow reverse
    local = yellow
    remote = green

[color "diff"]
    meta = yellow bold
    frag = magenta bold
    old = red bold
    new = green bold

[color "status"]
    added = yellow bold
    changed = green
    untracked = red
```

gibi ayar çekebilirsiniz. Yaptığımız ayarı kontrol etmek için;

```
git config color.status.added
# yellow bold
```

geldiyse ayar doğru yapılmış demektir.

`commit.template`

İlerleyen bölümlerde; Commit nedir? Commit Mesajı nedir? konusunda çok işimize yarayacak bir ayar özelliğidir. Commit mesajı yazarken bize hazır şablon çıkmasını sağlarız bu özellikle.

Şablon için bir dosya oluşturmak gerekiyor ve daha sonra kendi konfigürasyon dosyanızda bu dosyanın yerini bildirmeniz gerekiyor. Ben bu dosyayı `$HOME/.git-commit-template.txt` şeklinde kullanıyorum.

```
git config --global commit.template ~/.git-commit-template.txt # örnek
```

şeklinde konfigürasyonu ayarlamanız gerekir. Peki bu şablon dosyasında ne yazıyor?

```
[İlk 50 karakter: Özet]

[Açıklama: Neler oldu?]

[Fixed #]
```

gibi bir şema kullanabilirsiniz. Bu durumda her `git commit` dediğinizde bu şablon karşınıza çıkacak ve gerekli yerleri doldurmanız/silmeniz yeterli olacaktır.

`help.autocorrect`

Yanlış yazdığınızda, size yardımcı olacak bir asistan! Örneğin `git commit` yerine `git conmit` yazdığınızda bunun `commit` olduğunu anlayacak ve bu şekilde hareket edecek.

```
git config --global help.autocorrect true
```

Not: GIT pratiği açısından bunu aktif etmeyi önermiyorum. El alışkanlığı için doğru yazmayı öğrenmek her zaman daha doğru diye düşünüyorum.

`core.autocrlf`

İşletim sistemlerine göre **LINE ENDINGS** yani satır sonu/bitimi farklılıklar gösteriyor. Özellikle Windows! Bir projede hem Windows hem Unix kullanıcısı dosya kaydettiğinde, aralarında sıkıntı yaşamamak için ufak bir ayar yapmaları gerekiyor.

Windows kullanıcısı:

```
git config --global core.autocrlf true
```

Unix (*Mac/Linux vb...*)

```
git config --global core.autocrlf input
```

yapması her iki kullanıcı için de sıkıntıyı ortadan kaldırıyor. Windows satır sonlarına `\r\n` eklerken Unix türevleri `\n` ekliyor. `\n` **Line feed** anlamındadır ve sonraki satırı ifade eder. **C**arriage **R**eturn **L**ine **F**eed ise ek olarak cursor'u (*imleçi*) da başa alır. Bu bakımdan `\n` yerine `\r\n` kullanır.

Konfigürasyondaki bu ayarlama özelliği sayesinde Windows kullanıcısının düzenlendiği dosyayı Linux/Unix kullanıcısı açtığında sorun yaşamadığı gibi Windows kullanıcısı da Linux/Unix kullanıcısından gelen dosyaları sıkıntısız açabiliyor.

`core.whitespace`

GIT default olarak **boşluk** karakteriyle ilgili bir kısım anlama özelliği ile geliyor. Default olarak gelen **3 seçenek**;

1. **blank-at-eol**: Satır sonundaki space (*boşluk*) karakter(ler)i
2. **blank-at-eof**: Dosya sonundaki space (*boşluk*) karakter(ler)i
3. **space-before-tab**: Satır başındaki `tab` karakterinden önce gelen space (*boşluk*) karakter(ler)i

Buna ek olarak kapalı (*disabled*) gelen **3 özellik**;

1. **indent-with-non-tab**: `tab` yerine `space` ile indent (girintilenmiş) edilmişler
2. **tab-in-indent**: Satır içindeki `tab` ile girintilenmiş yerler
3. **cr-at-eol**: Satır sonundaki **carriage returns** yani ENTER/RETURN tuşuna basınca çıkanlar düzgün mü?

Bu konu ile ilgili farklı konfigürasyon kullanımları bulunmakta. **whitespace** sorunu yaşamamak için ben;

```
git config --global core.whitespace fix,-indent-with-non-tab,trailing-space,cr-at-eol
```

şeklinde kullanıyorum. Başka bir örnekte:

```
git config --global core.whitespace trailing-space,space-before-tab
```

`core.excludesfile`

`.gitignore` dosyasının global olanı. Dizin bazında `.gitignore` ile dosyaları revizyon kontrol dışında tutabiliyoruz. Bunu genel olarak kullanmak istersek, aynı `~/.gitconfig` gibi `~/.global_gitignore` gibi bir dosya oluşturup:

```
git config --global core.excludesfile ~/.global_gitignore
```

gibi ayarlarsak, otomatik olarak belirlediğimiz dosyaları revizyon kontrol dışına alabiliriz. Unutmayın ki, proje bazlı `.gitignore` dosyası, repository içinde olduğu için, projeye katkı yapan diğer kullanıcılar tarafından da kullanılabiliyor.

Bu bakımdan, global olarak **ignore** ettiğiniz şeyler başka kimseye geçmeyeceği için dikkatli kullanın.

Örnek bir `.gitignore` :

```
.DS_Store
._*
.Spotlight-V100
.Trashes
*.pyc
xcuserdata
.sass-cache
.bundle
vendor/bundle
```

Pek çok temporary (*temp*) dosya ve benzeri şeyleri ignore eden bu dosyaya bir göz atın derim.

`rerere`

Reuse **re**corded **re**solution yani merge esnasında yaşanan conflict'lerin çözümlerini hatırla ve bir daha aynısı olursa otomatik olarak çöz! GIT bazen bizden daha akıllı olabiliyor :) Bu özelliği aktive etmek için;

```
git config --global rerere.enabled true
```

yapmak yeterli.

`status.submoduleSummary`

Submodule'ler le çalıştığınız zaman işinize yarayacak. `git status` dediğinizde, repo altındaki submodule'lerin de durumunu görmeniz gerekebilir.

```
git config --global status.submoduleSummary true
```

Örnek Konfigürasyon Dosyası

Bu örneği kendinize göre düzenleyip `~/.gitconfig` olarak kullanabilirsiniz.

```
[user]
    name = AD SOYAD
    email = E-POSTA
[commit]
    template = ~/.git-commit-template
[alias]
    add-modified            = !"git status -sb | grep '^ M ' | sed 's/ M //' | xargs g
it add"
    add-untracked           = !"git status -sb | grep '^??' | sed 's/?? //' | sed 's/.
*/\"&\"/' | xargs git add"
    branches                = for-each-ref --sort=-committerdate --format='%(color:whi
te)[%(refname:short):%(color:yellow)%(objectname:short)%(color:reset)]%(color:reset) \
t %(color:red)%(authorname)%(color:reset) \t %(color:blue)%(authordate:relative)%(colo
r:reset)' refs/remotes
    show-ignored            = "ls-files -o -i --exclude-standard"
    show-todays-diff        = "diff --shortstat '@{0 day ago}'"
    show-untracked          = "ls-files --others --exclude-standard"
    add-deleted             = "add -u"
    root-commit             = "rev-list --max-parents=0 HEAD"
    next-commit             = !"git checkout $(git log --reverse --ancestry-path --pre
tty=%H HEAD..master | head -1)"
    unstage                 = "reset HEAD --"
    uncommit                = "reset --soft HEAD^"
    wip                     = !"git add -A; git ls-files --deleted -z | xargs -0 git r
m; git commit -m 'wip'"
    initial-commit-tr       = "commit --allow-empty -m'[root] İlk commit'"
    initial-commit          = "commit --allow-empty -m'[root] Initial commit'"
    br                      = "branch -v"
    bra                     = "branch -avv"
    brd                     = "branch -d"
    brm                     = "branch --no-mergedd"
    brnm                    = "branch --no-merged"
    brr                     = "branch -rv"
    ci                      = "commit"
    co                      = "checkout"
    df                      = "diff --word-diff"
    dfn                     = "diff --name-only"
    fc                      = "commit --allow-empty -m"
    lg                      = "log --graph --decorate --oneline --all"
    lg2                     = "log --graph --decorate --pretty='%C(auto)%h %G?%d %C(wh
ite)%s%C(reset) [%aE, %ad]' --date=relative"
    lgs                     = "log --graph --decorate --oneline"
    list-remote-tags        = "ls-remote --tags"
    ls                      = "ls-files"
    pullr                   = "pull --rebase"
```

```
    rmt                     = "remote -v"
    st                      = "status"
    stu                     = "status --untracked-files"
    sti                     = "status --ignored"
    sts                     = "status -sb"
    track-origin-master     = "branch --set-upstream-to=origin/master master"
[color]
    ui = auto
[color "branch"]
    current = yellow reverse
    local = yellow
    remote = green
[color "diff"]
    meta = yellow bold
    frag = magenta bold
    old = red bold
    new = green bold
[color "status"]
    added = yellow bold
    changed = green
    untracked = red
[core]
    excludesfile = ~/.gitignore
    whitespace = fix,-indent-with-non-tab,trailing-space,cr-at-eol
    pager = less -FRX
    autocrlf = input
    safecrlf = true
    disambiguate = commit
    abbrev = 12
[push]
    default = tracking
[filter "media"]
    clean = git-media-clean %f
    smudge = git-media-smudge %f
[pull]
    rebase = true
[fetch]
    prune = true
[rerere]
    enabled = true
[help]
    autocorrect = 0
[diff]
    compactionHeuristic = 1
[status]
    submoduleSummary = true
```

Kısa Yollar: alias

Günlük hayatta sık kullandığımız komutlar için kısa yollar oluşturabiliriz. Buna `alias` deniyor. Örneğin;

```
git status
```

çok kullanacağımız bir işlem olacak. Bu bakımdan her seferinde `git status` yazmak yerine `git st` ya da `git s` gibi `alias` tanıtmak mümkün. Aynı diğer konfigürasyon elementleri gibi:

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

ya da `~/.gitconfig` dosyanızı açıp:

```
[alias]
    st = status
```

gibi düzenleme yapabilirsiniz. Eğer alias'ın başında `!` işareti olursa bu **shell komut**u çalıştırabileceğimiz anlamına gelir. Örneğin takip altında olmayan dosyaları otomatik olarak takibe almak için komut satırından şu işlemler serisini yapabiliriz:

```
git status -s

# iki tane yeni takip altında olmayan dosya olduğunu düşününelim, çıktısı
# aşağıdaki gibi olsa;

 M index.html
?? test.jpg
?? global.js
```

`M` modified anlamında yani daha önce kontrol altına alınmış, `??` ise un-tracked anlamında, yani yeni dosya, hiçbir revizyon bilgisi yok. Şimdi;

```
git status -s | grep -e "^??"

# sonuç

?? test.jpg
?? global.js

git status -s | grep -e "^??" | awk '{ print $2 }'

# sonuç
test.jpg
global.js
```

ve mini Bash Script Crash-Course finalinde;

```
git status -s | grep -e "^??" | awk '{ print $2 }' | xargs git add
```

yaparak sadece un-tracked olanları ekledik. Gördüğünüz gibi yaklaşık 4 komutu zincirleme çağırdık. Bunu kolay yoldan yapmak için `alias` olarak ekleyebiliriz:

```
git config --global alias.add-untracked '!git status -sb | grep -e "^??" | awk "{ prin
t \$2 }" | xargs git add'
```

Bu işlemden sonra `git add-untracked` dediğimizde, yeni oluşan, revizyon kontrol altında olmayan dosyalar otomatik olarak eklenecektir. Diğer örnek kısa yollar için Örnek Konfigürasyon Dosyası incelenebilir.

Commit Nedir?

GIT, kendi içinde özel bir GRAPH yapısı kullanıyor. Bunun adı: **Directed Acyclic Graph**. Bence biraz korkutucu :) Bu nedir diyen varsa detayları linkten öğrenebilir. Kabaca ortada bir ağaç yapısı var. Ağacın kolları var, dalları var. Kendi özel yapısı içinde GIT değişiklikleri kendi yöntemleriyle saklar.

Akla gelebilecek en basit yöntem *delta-diff* yani sadece değişen şeyleri saklamak yerine GIT komple o anın fotoğrafını çeker. Bu aslında o an'ın **snapshot**'ı dır ve GIT buna **Commit** der.

Commit yaptığınız zaman GIT, adı **commit-object** olan bir taşıyıcı saklar. Bu taşıyıcı içinde **stage** edilmiş içerik, commit'i yapan kişi bilgileri, varsa bağlı olduğu bir üst commit ya da **branch**'lerin **merge** edilmesi (*birleştirilmesi*) sonunda oluşmuş bir commit ise birden fazla branch bilgisi saklar. Bunlar aslında birer işaretçi yani **pointer**'dır.

Commit, içinde hangi **tree** yapısına dahil olduğu bilgisini de saklar. İlk commit dışındaki tüm commit'lerin bir **parent-commit**'i bulunur. Sıfır bir repo içinde yapılan ilk commit aslında o repo'nun **root-commit**'idir. Hiçbir commit'ten türememiştir.

Örnek bir log çıktısına bakalım:

```
* b72ce45cafdc (HEAD -> master) fixed grammar at license section
* 504714498c31 hooks folder deleted
* 49261317ef93 Release: v0.1.0
* c0672c7d5be6 Ready for v0.1.0
* 71ff5c9dfac2 Added installation feature.
* 9bf387240b22 Changed message storage file.
* 1e7071c6e6fe Ready to release. Need to finish README.
* c3fd828979bc added: README file
* b1d90b39ba10 [root] initial commit
```

Son yapılan commit: **b72ce45cafdc**. Bu commit'in parent'ı **504714498c31** ve 504714498c31 commit'in parent'ı **49261317ef93**. Parent'ı olmayan tek commit: **b1d90b39ba10**

Genelde şematik olarak gösterilirken;

```
                                master
                                   |
49261317ef93 <- 504714498c31 <- b72ce45cafdc (son commit)
```

zaman çizelgesi soldan-sağa akarken commit'lerin ilişkisi tam ters şekildedir. Her zaman kim kimin üstü altı (*parent/child*) durumu önemlidir.

İlk Commit

Dedik ya, zamanın, o anın fotoğrafını çekiyoruz... Fotoğrafı çekebilmek için repomuzda dosyaların olması gerekiyor. Ben, yeni bir projeye başladığım zaman, ilk yaptığım commit'i **boş commit** olarak yapıyorum.

Reponun ilk commit'i aslında root-commit olup hiçbir parent commit'e sahip değildi. Madem öyle, hiçbir dosya ile de ilişkilendirmek istemiyorum:

```
$ cd /tmp/
$ git init my-awesome-tool && cd $_
$ git commit --allow-empty -m '[root] Initial commit'
[master (root-commit) 2a88b16f848c] [root] Initial commit

$ git log --oneline
* 2a88b16f848c (HEAD -> master) [root] Initial commit
```

Commit mesajlarını **İngilizce** yazacağım. Elimizi buna alıştırmamız lazım. Hep açık-kaynak projelerde çalışırken ya da ekibe Türkçe bilmeyen biri dahil olduğunda ya da size yabancı bir ekiple / projeyle çalıştığınızda sıkıntı çekmemeniz için iyi bir antrenman olur :)

Commit Mesajı Nedir?

GIT'in resmi dokümantasyon sitesinde commit'in açıklamasını yaparken, mesaj ile ilgili kısımda;

> ... with a log message from the user describing the changes.

der. Yani kullanıcının yaptığı değişiklikleri tarif ettiği, anlattığı kayıttır bu aslında. Çünkü GIT değişiklikleri takip eden bir araçtır. Teknik olarak GIT deşikliğin ne olduğunu biliyor ama sonuç olarak kodu yazan insan olduğu için, insanın anlayacağı şekilde açıklamak gerekiyor nelerin değiştiğini.

En hızlı şekilde commit mesajı yazmak için günlük hayatta;

```
git commit -m "MESAJ"
```

kullanırız. Bu aslında kısa mesaj kullanımıdır. Örneğin `git log --oneline` kullanımında aldığımız çıktı:

```
a58834812b45 (HEAD -> master, origin/master, origin/HEAD) Merge pull request #1295 fro
m ninoseki/master
875f36f963dd Change relative URLs to absolute URLS
a1b0de7b45a7 Fixed broken links in rack-protection/README.md
ec08e37bf8b4 Linkify protection docs
ea1a21c0734e Update homepages for sinatra-contrib and rack-protection
```

gibidir. `COMMIT_ID` ve `COMMIT_MESSAGE` şeklinde görürüz. Bu kısa mesaj, aslında commit mesajının ilk satırıdır ve bu ilk satır **50 karakter**dir. ID'si a58834812b45 olan commit'e bakalım:

```
git show a58834812b45

Merge pull request #1295 from ninoseki/master

Fixed broken links in rack-protection/README.md
```

İlk satırda kısa açıklama, bir satır boşluk ve mesajın devamı bulunur. Ben de dahil olmak üzere, pek çok geliştirici, yaptığı değişiklikleri uzun uzun yazmak yerine ışık hızıyla;

```
git commit -m 'up'
git commit -m 'update'
git commit -m 'wip'
```

gibi, aslında kavgada bile yapılmayacak hareketleri yapmışızdır. Neden? Kim onca değişikliği oturup yazacak şimdi? İşte en azından bu durumları minimal düzeye indirmek için tavsiye edilen ilk yöntem: `git commit` ile commit yapmak, `git commit -m 'MESAJ'` kullanmamak!

Bu yöntemle karşımıza `git config core.editor` ile belirlenmiş text editörü çıkar ve ferah fezah mesaj yazacak bir alanla karşı karşıya kalırız. Böyle bir durumda otomatikleştirilmiş şablon mesaj çok işimize yarayabilir.

İyi Bir Commit Mesajı Nasıl Olmalı?

Bence temel kurallar;

1. İlk satırda (*50 karakteri geçmeden*) özet bilgi vermek
2. Detayları **bir boş satır bırakıp** alta yazmak
3. Mutlaka yapılan değişiklileri iyi izah etmek

Şu mesaj örneğine bakalım:

```
Sitemizin yeni bölümü için index sayfası oluşturuldu.

Yaptığımız projede, ABCD bölümü için yeni statik bir html sayfası
gerekiyordu, bunun için index.html dosyasını oluşturdum ve
front-end'i yazacak arkadaşlara gerekli bilgiyi verdim.

Yeni bir paragraftaı aynı markdown kullanır gibi kullandım.

    - Bu şekilde list item yaptım,
    - Listeye devam ettim

Bu konu ile ilgili ticket'lar:

- http://example.com/ticket/1
- http://example.com/ticket/2

# 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:   index.html
#
```

Kısa log'a bakan kullanıcı;

```
$ git log --oneline

8833c9cfc Sitemizin yeni bölümü için index sayfası oluşturuldu.
:
:
```

şeklinde görecek. Gayet açıklayıcı olduğunu düşünüyorum. Detayları merak eden olursa yukarıda yazan hikayeyi okuyabilir, hangi issue/ticket ile alakalı olduğunu görebilir.

Mesajı içinde `#` ile başlayan satırlar yorum satırlarıdır. Mesaja dahil olmazlar. `git commit` dedikten sonra, gelen ekranda hiçbir şey yazmadan çıkarsanız **commit yapmaktan vazgeçmiş** olursunuz:

```
Aborting commit due to empty commit message.
```

mesajını alırsınız. Yani **boş mesaj** olamaz! (*Bazı özel durumlar dışında...*)

Basit Kullanım Örneği

GIT **146**'dan fazla komut içermektedir. Bunların bazıları günde sadece 1-2 kere, bazıları 40-50 kere, bazıları da ayda-yılda belki 1 kere kullanacağınız komutlar olacaktır. Eğer;

```
git help everyday
```

derseniz, karşınıza bir geliştiricinin ortalama kullanacağı komutları, nasıl kullanacağı bilgisini ve günlük rutin işlerinizde size yardımcı olabilecek çalışma yöntemlerini gösteren harika bir yardım sayfası gelir.

Kabaca bakıldığında;

1. `git init`
2. `git log`
3. `git status`
4. `git checkout`
5. `git add`
6. `git reset`
7. `git commit`
8. `git pull`
9. `git push`

en sık kullanacağınız komutlardan olacaktır. Hemen sıfırdan yeni bir repo oluşturalım:

```
$ cd /tmp/
$ git init git-basics && cd $_
Initialized empty Git repository in /private/tmp/git-basics/.git/

$ git commit --allow-empty -m'[root] Initial commit'
[master (root-commit) 7639a730f5c7] [root] Initial commit
```

Şimdi içine bir dosya atalım ve reponun durumuna bakalım: `git status`

```
$ touch README.md
$ git status

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

    README.md

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

Takip dışında dosyalar var! (*Untracked files*) Eğer bu dosyaları takip altına almak isterseniz: `git add <file>` yapın... Peki yapalım: `git add`

```
$ git add README.md
$ git status

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

    new file:   README.md
```

GIT, yeni bir dosya takip etmeye başladığını anladı. Çünkü biz `git add` ile emir verdik GIT'e. Dikkat ettiyseniz şöyle bir uyarı var:

```
(use "git reset HEAD <file>..." to unstage)
```

`git add README.md` dediğimiz anda, `README.md` dosyasını **STAGING AREA** yani sahneye almış olduk. Sahneye alınan her dosya commit edilebilir hale gelir. Bu dosyanın içeriği GIT'in **index** mekanizmasına eklenmiştir. Bu işlem **STAGING** olarak da bilinir.

Eğer henüz bu dosyayı takibe almak istemiyor ya da değişikliği geri almak istiyorsanız `git reset HEAD README.md` yapıp sahneden aşağı indirmeniz yani **UNSTAGE** etmeniz gerekir.

Bu noktada iki yeni kavram çıkar karşımıza. `HEAD` ve `git reset`.

HEAD

`HEAD` aslında bir kısayol yani alias'dır. Bulunduğunuz branch'deki en yeni / en son commit'i işaret eder.

Şimdi örneğe dönüp hiçbir commit yapmadan `git log --oneline` dersek:

```
* 7639a730f5c7 (HEAD -> master) [root] Initial commit
```

`HEAD`'in **master** branch'de commit id'si **7639a730f5c7** olanı işaret ettiğini görürsünüz. Haydi yeni bir şey daha...

COMMIT ID

GIT, mutlaka her commit'e dünyada eşi benzeri olmayan bir **ID** verir. Aslında **7639a730f5c7** diye gördüğümüz şey; **7639a730f5c7979ca8a5ecaed3731e0e360f280a** sayısının ilk 12 karakteridir. Bu kısa haline **Short-SHA1** (*Kısa SHA1*) denir. Kısa SHA1'in uzun halini bulmak için;

```
git rev-parse 7639a730f5c7 # 12 karater
git rev-parse 7639a7       # 6 karakter
```

yapmak yeterlidir. Uzun SHA1'in ilk 6 ya da 7 karakteri de iş görür... COMMIT ID aynı zamanda o anı ifade ettiği için ilgili revizyonu da ifade eder.

Her commit'in tekil olması zorunluluğu, GIT'in dağıtık çalışması için zorunludur. Bir takımda **N tane kişi** aynı dosyalarla çalışacak ve hepsinde projenin bir kopyası olacak.

Keza hiçbiri de internete bağlı olma zorunluluğu olmadan çalışacak. Bu bakımdan her yapılan commit'in **başka commit'lerle karışmaması** ve unique olması gerekiyor.

İşte GIT'in en ayrıştırıcı özelliği de bu.

Tekrar örneğe dönelim. GIT bize bilgi verdi ya, eğer istersek `git reset` yaparız. İleriki bölümlerde daha detaylı değineceğim ama şimdi yeri gelmişken hızlıca açıklamaya çalışayım. `git reset REVİZYON DOSYA` kullanım şekillerinden biridir. `HEAD` neyi işaret ediyordu? **7639a730f5c7** numaralı commit'i. Bu revizyonda `README.md` diye bir dosya var mıydı?

```
$ git reset HEAD README.md
$ git status

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

    README.md

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

Hayır, takip altında böyle bir dosya yoktu. Tekrar başladığımız yere döndük... Şimdi tekrar ekleme işini yapalım ve ardından da ilk commit'i yapalım:

```
$ git add README.md
$ git commit -m 'Added: readme file'

[master 6601df828690] Added: readme file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

Commit başarıyla gerçekleşti. GIT yeni **unique** (*benzersiz*) bir ID verdi, **6601df828690** ek olarak 1 dosyanın eklendiğini, dosya içinde metinsel herhangi bir değişiklik olmadığını söyledi. `0 insertions(+), 0 deletions(-)`

Peki şimdi durum ne?

```
$ git status

On branch master
nothing to commit, working tree clean
```

master branch'deyiz, her şey yolunda!. `git add` kullanırken;

```
# git add <DOSYA>
$ git add README.md

# git add <DİZİN>
$ git add images/

# git add .
# bulunduğun dizindeki her şeyi
# . unix ifadesidir ve current working directory'i ifade eder.
$ git add .

# git add *.py
# foo.py import.py run.py gibi sonu .py ile biten dosyaları ekler
$ git add *.py

# git add test-*.txt
# test-foo1.txt test-foo2.txt test-hello-world.txt gibi
$ git add test-*.txt
```

yapmak mümkün. İşlem sırası olarak önce **add** sonra **commit** yaptık. Bu süreci tek harekete indirmek mümkün:

```
$ git commit -a -m 'Added: all untracked files at once'
```

`commit -a` ile tüm **untracked** (*henüz takibe alınmamış*) olan dosya/dizin ne varsa **staging**'e at diyoruz. `-m` de tabiki commit mesajı. Ben bu yöntemi neredeyse hiç kullanmıyorum. Mutlaka ilgili dosyaları ekleyip daha sonra commit ediyorum.

Üç Aşamalı Dosya Sistemi

GIT ile çalışırken 3 aşamalı dosya sistemini hep aklımızda tutmalıyız:

1. Staged
2. Modified
3. Commited

Staged

Bazı değişiklikler ya da eklemeler yaptın, sepete attın ama henüz GIT'e bildirmedin! Yani sahneye aldın fakat öylece duruyor orada.

Modified

Değişiklikler var, GIT bunun farkında ama henüz kaydetmedin yani commit etmedin! Bu ne demek? Hemen bakalım:

```
$ ls

README.md
file-info-1.txt
file-info-2.txt
file1.txt
file2.txt
```

Şimdi `file-info-2.txt` üzerinde değişiklik yapalım.

```
$ echo 'adding another great line' >> file-info-2.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:   file-info-2.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

GIT, değişikliği anladı, bu dosyanın değiştiğini yani **modified** olduğunu söylüyor. Değişikliği geri almak için `git checkout -- <file>` yapabiliriz. Acaba `file-info-2.txt` içinde ne yazıyor?

```
$ cat file-info-2.txt

file information 2
adding another great line
```

Hmmm... peki alalım geri:

```
$ git checkout -- file-info-2.txt
$ git status

On branch master
nothing to commit, working tree clean
```

Dosyanın içeriği?

```
$ cat file-info-2.txt
file information 2
```

Şimdi biraz daha değişik bir şey deneyelim. Tekrar aynı işlemi yapalım ama küçük bir farkla:

```
$ echo 'adding another great line' >> file-info-2.txt
$ git add file-info-2.txt
$ echo 'adding another great line for the 3rd time' >> file-info-2.txt
$ git status

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

    modified:   file-info-2.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:   file-info-2.txt
```

Durum şu; `file-info-2.txt` dosyası sanki 2 kere **modified** yani değişikliği uğramış ama birinde **staged**, diğerinde ise **modified**. Evet aynen de böyle oldu. Değişiklik yaptık ve `git add` ile staging'e aldık, commit yapmadık ve tekrar değişiklik yaptık...

Tekrar ekleyelim ki bu ikinci değişiklik de kayıt altına alınsın:

```
$ git add file-info-2.txt
$ git status

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

    modified:   file-info-2.txt
```

Committed

Veri, yerel veritabanına güvenli olarak kaydedildi. Neticede GIT kendi yerelinde, kendi anlayacağı şekilde bir veritabanı saklıyor. Şimdi yukarıda yarım kalan işi bitirelim:

```
$ git commit -m 'staged, modifed and commited example'

[master 5fe19937b38f] staged, modifed and commited example
 1 file changed, 2 insertions(+)

$ git log --oneline

5fe19937b38f staged, modifed and commited example
5579b829b6a8 file added via git add -p
c52805a3c8f5 added two info files for demo purposes
bb965e45b9ca Both files are added via git add -i
6601df828690 Added: readme file
7639a730f5c7 [root] Initial commit
```

İnteraktif Ekleme: `git add -i`

Eğer `git add -i` derseniz, **Interactive Mode**'a geçersiniz. Bu durumda karşınıza çeşitli seçenekler çıkar:

```
$ touch file{1,2}.txt
$ ls -al

total 0
drwxr-xr-x   6 vigo  wheel   204 Jun 25 19:48 .
drwxrwxrwt  59 root  wheel  2006 Jun 25 19:39 ..
drwxr-xr-x  14 vigo  wheel   476 Jun 25 19:48 .git
-rw-r--r--   1 vigo  wheel     0 Jun 25 18:53 README.md
-rw-r--r--   1 vigo  wheel     0 Jun 25 19:48 file1.txt
-rw-r--r--   1 vigo  wheel     0 Jun 25 19:48 file2.txt

$ git status

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

    file1.txt
    file2.txt

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

$ git add -i

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch      6: diff      7: quit      8: help
What now>
```

Evet, şimdi ne yapacağız? **4** yani takip dışındakileri ekleyelim.

```
  1: file1.txt
  2: file2.txt
Add untracked>>
```

Bu durumda ya `1,2` yazıp iki dosyayı da ekleyeceğiz ya da teker teker ilerleyeceğiz. Ben `1,2` yazdım:

```
* 1: file1.txt
* 2: file2.txt
Add untracked>>
```

`*` bunları işleme girdiğini belirtiyor. Tekrar `<ENTER>` yapıp ilerliyorum:

```
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>
```

İşim bitti, `7` ile çıkıyorum ve duruma bakıyorum: `git status` :

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

    new file:   file1.txt
    new file:   file2.txt
```

Şimdi commit zamanı...

```
$ git commit -m 'Both files are added via git add -i'
[master bb965e45b9ca] Both files are added via git add -i
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

İnteraktif modda, oldu ya son anda dosyayı eklemekten vazgeçtiniz;

```
$ touch file3.txt && git add -i

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch      6: diff      7: quit      8: help
What now> 4
  1: file3.txt
Add untracked>> 1
* 1: file3.txt
Add untracked>>
```

`-` ile çıkartabilirsiniz. `-1` ya da `-INDEKS_NO` .

```
Add untracked>> -1
  1: file3.txt
Add untracked>> <ENTER>
Add untracked>>
No untracked files.

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch      6: diff      7: quit      8: help
What now> 7
Bye.
```

Patch Mode'da Ekleme: `git add -p`

Belkide GIT'in yapılma sebeplerinden biri **Patching**. Linus Torvalds, Linux kernel'i geliştirirken, diğer katkı yapan developerlar, yaptıklarını e-posta ile patch formatında yolluyorlarmış.

Bu e-postalarla mücadele etmek, gelenleri kontrol etmek, bir tür **review** işleminden geçirmek vs çok zahmetli işler. GIT bu durum için özel bir seçeneğe sahip.

```
$ echo 'file information 1' > file-info-1.txt
$ echo 'file information 2' > file-info-2.txt
$ ls
README.md
file-info-1.txt
file-info-2.txt
file1.txt
file2.txt

$ git add .
$ git status

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

    new file:   file-info-1.txt
    new file:   file-info-2.txt

$ git commit -m 'added two info files for demo purposes'
```

Şimdi, `file-info-1.txt` dosyasına bir-kaç satır ekleyelim:

```
$ echo 'added new line' >> file-info-1.txt
$ echo 'added one more new line' >> file-info-1.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:   file-info-1.txt

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

Şimdi patch mode'a geçelim:

```
$ git add -p

diff --git a/file-info-1.txt b/file-info-1.txt
index 833822e8a04b..a333d502478e 100644
--- a/file-info-1.txt
+++ b/file-info-1.txt
@@ -1 +1,3 @@
 file information 1
+added new line
+added one more new line
Stage this hunk [y,n,q,a,d,/,e,?]?
```

GIT, bildiği son hali ile yeni değişikliğin arasını bize `diff` ile gösteriyor ve soruyor. Bu **hunk**'ı yani parçayı ne yapayım?

- `y` : YES, bu hunk'ı al.
- `n` : NO, bu hunk'ı alma.
- `q` : QUIT, hiçbir şey yapmadan çık ve devam etme.
- `a` : ALL, bu hunk dahil, dosyadaki diğer tüm hunk'ları al.
- `d` : DON'T, bu hunk dahil, diğer tüm hunk'ları alma.
- `/` : SEARCH, girilecek REGEX paternine göre hunk ara.
- `e` : EDIT, elle hunk'ı düzenle.
- `?` : HELP, yardım için.

Ben `a` yapıyorum ve `git status` :

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

    modified:   file-info-1.txt
```

Bu yöntem kodu tekrar gözden geçirme imkanı verdiği gibi, son dakikada bir şeyleri düzeltme değiştirme şansı da veriyor.

Bu patch modu aslında çaktırmadan `git add -i` yapıyor ve direk olarak oradaki patch seçeneğini açıyor. Daha fazla bilgi almak için `git help add` demek mümkün.

Repo'nun Durumu: `git status`

Bu noktaya kadar pek çok kez `git status` yaptık. Ek birkaç parametre ile değişik çıktılar almak mümkün:

`git status -s` ve `git status -sb`

Short Status yani kısaca durum bilgisi için kullanılır. `-b` bulunduğun branch'i de göster anlamındadır:

```
$ git status -s

 M file1.txt
AM new-file-1.txt
A  new-file-2.txt
?? new-file-3.txt

$ git status -sb

## master
 M file1.txt
AM new-file-1.txt
A  new-file-2.txt
?? new-file-3.txt
```

Bu tek harflerin bir anlamı var.

- ` ` = unmodified, boşluk karakteri, değiştirilmemiş
- `M` = modified, değişiklik var
- `A` = added, eklendi yani staged
- `D` = deleted, silindi
- `R` = renamed, dosya adı değişti
- `C` = copied, kopyalandı
- `U` = updated but unmerged, index güncellendi ama merge edilmedi

`git status --ignored`

Eğer exclude edilmiş, yani tanımlanan dosyalar, dizinler, ya da dosya türlerini GIT takibe almasın demişsek, varsa bu tür dosyalar, status içinde bunları da göster demektir.

Dosyaları nasıl takip dışında bırakırız?

`git status --untracked-files`

Örnek projede, `git status` dediğimiz zaman;

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

    new file:   new-file-1.txt
    new file:   new-file-2.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:   file1.txt
    modified:   new-file-1.txt

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

    new-file-3.txt
    test-folder/
```

görüyoruz. Bizi ilgilendiren kısım **Untracked files:** kısmı yani henüz takip altına alınmamış dosyalar. Bir dosya bir de dizin görüyorum. Acaba bu dizin altında başka dosya var mı?

```
$ git status --untracked-files

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

    new file:   new-file-1.txt
    new file:   new-file-2.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:   file1.txt
    modified:   new-file-1.txt

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

    new-file-3.txt
    test-folder/test-file-a.txt
    test-folder/test-file-b.txt
    test-folder/test-file-c.txt
```

Tam üç tane dosya varmış!...

Log’a Bakış

Üzerinden hassasiyetle durduğumuz **Commit Mesajı** konusu işte bu kısımda çok önemli bir rol oynar çünkü o yazdığımız mesajlar, repomuzun tarihçesini gösterir bize.

`git log` dediğimizde;

```
commit 5579b829b6a801f8ebd7597db7e1a9fd43f28d49
Author: Uğur Özyılmazel <ugurozyilmazel@gmail.com>
Date:   Sun Jun 25 20:35:42 2017 +0300

    file added via git add -p

commit c52805a3c8f5c78588e9b7fbc2e4f1b31675d5e2
Author: Uğur Özyılmazel <ugurozyilmazel@gmail.com>
Date:   Sun Jun 25 20:19:43 2017 +0300

    added two info files for demo purposes

commit bb965e45b9ca8d60de77ff066f6f4eb6ea819c97
Author: Uğur Özyılmazel <ugurozyilmazel@gmail.com>
Date:   Sun Jun 25 19:55:24 2017 +0300

    Both files are added via git add -i

commit 6601df8286905fee9942dfc5fe6a36b7e95f1e7e
Author: Uğur Özyılmazel <ugurozyilmazel@gmail.com>
Date:   Sun Jun 25 19:19:40 2017 +0300

    Added: readme file

commit 7639a730f5c7979ca8a5ecaed3731e0e360f280a
Author: Uğur Özyılmazel <ugurozyilmazel@gmail.com>
Date:   Sun Jun 25 18:53:15 2017 +0300

    [root] Initial commit
```

gibi bir çıktı ile karşılaşırız. Burada, default olarak sondan başa doğru sıralanmış bir şekilde, yapılan commit, yapan, yapılış tarihi ve mesajı gibi meta bilgilerini görüntüleriz.

Alacağı farklı parametrelerle çok daha öz ve kolay anlaşılır bilgiler verir bize `git log` :

```
$ git log --oneline

5579b829b6a8 file added via git add -p
c52805a3c8f5 added two info files for demo purposes
bb965e45b9ca Both files are added via git add -i
6601df828690 Added: readme file
7639a730f5c7 [root] Initial commit

$ git log --oneline --stat

5579b829b6a8 file added via git add -p
 file-info-1.txt | 2 ++
 1 file changed, 2 insertions(+)
c52805a3c8f5 added two info files for demo purposes
 file-info-1.txt | 1 +
 file-info-2.txt | 1 +
 2 files changed, 2 insertions(+)
bb965e45b9ca Both files are added via git add -i
 file1.txt | 0
 file2.txt | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
6601df828690 Added: readme file
 README.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
7639a730f5c7 [root] Initial commit
```

Benim bir alias'ım var. `lg` yani `git lg` olarak çağırıyorum:

```
$ git log --graph --decorate --oneline --all # git lg

* 5579b829b6a8 (HEAD -> master) file added via git add -p
* c52805a3c8f5 added two info files for demo purposes
* bb965e45b9ca Both files are added via git add -i
* 6601df828690 Added: readme file
* 7639a730f5c7 [root] Initial commit
```

Siz de yapabilirsiniz:

```
$ git config --global alias.lg "log --graph --decorate --oneline --all"
$ git help lg
`git lg' is aliased to `log --graph --decorate --oneline --all'
```

Bazı Dosyaları Takip Etmemek: `.gitignore`

Bazı durumlarda dosya ya da dosyaları ya da dizinleri takip dışında tutmak istersiniz. Örneğin uygulamanızın `log` dosyaları, ya da kullandığınız text editörü ile ilgili sadece sizi ilgilendiren dosyaları izole etmek isteyebilirsiniz.

Belki de kullanıcı adı, şifre, ssh bilgileri ya da **SECRETS** dediğimiz `ENVIRONMENT` değişkenlerinin ortalıkta dolaşmaması gerekebilir.

İşte bu tür durumlarda GIT'e bu dosyaları kaydetmemesini, yani takip etmemesini söyleriz. Bu işleme **ignore** işlemi yani görmezden gelme işlemi denir.

Aynı konfigürasyon dosya mantığında, ya proje bazlı **local** ignore dosyaları kullanırsınız ya da kendi `~/.gitconfig` dosyasında tanımladığınız `excludesfile` değişkenindeki direktifleri kullanırsınız.

```
[core]
    ;
    excludesfile = ~/.gitignore
    ;
```

Ya da dizin bazında işlem yapabilirsiniz. Repo'nun root'unda duran `.gitignore` dosyası en derindeki dizine kadar etki eder. Eğer isterseniz alt derinlikteki dizinlerde başka tanımlamalar yapabilirsiniz. Yani şöyle bir repo olsa:

```
.
├── sub-folder
│   ├── demo.txt
│   └── demo.xyz
├── file-1.txt
├── file-2.txt
├── file-3.txt
├── file-4.txt
└── file.xyz
```

Hemen kontrol edelim durum nedir?

```
$ git status --untracked-files

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

    file-1.txt
    file-2.txt
    file-3.txt
    file-4.txt
    file.xyz
    sub-folder/demo.txt
    sub-folder/demo.xyz

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

Tüm dosyalar **untracked** yani eklenmeyi bekliyor. `xyz` extension'ı olan dosyaları revizyon kontrol dışında tutalım. Bunun için projenin root'una `.gitignore` dosyası ekliyorum:

```
$ touch .gitignore
$ echo '*.xyz' >> .gitignore
$ git status --untracked-files

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

    .gitignore
    file-1.txt
    file-2.txt
    file-3.txt
    file-4.txt
    sub-folder/demo.txt

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

Dikkat ettiyseniz artık `file.xyz` , `sub-folder/demo.xyz` dosyaları **untracked** olarak görünmüyor. Özellikle `git status --untracked-files` şeklinde kullandım ki alt dizinlerin altındaki dosyaların da durumunu görebilelim.

Hatta hangi dosyaların **ignore** edildiğini;

```
$ git ls-files -o -i --exclude-standard

file.xyz
sub-folder/demo.xyz
```

şeklinde de görebiliriz. Ben olsam bunu bir **alias** yapardım :) Şimdi dizin bazlı izolasyonu görelim.

```
$ mkdir no-more-php-files
$ touch test.php
$ touch no-more-php-files/demo.php
$ touch no-more-php-files/test.rb
$ tree
.
├── no-more-php-files
│   ├── demo.php
│   └── test.rb
├── sub-folder
│   ├── demo.txt
│   └── demo.xyz
├── file-1.txt
├── file-2.txt
├── file-3.txt
├── file-4.txt
├── file.xyz
└── test.php
```

`no-more-php-files` dizini altındaki `*.php` dosylarını takip dışı bırakmak istiyorum:

```
$ echo '*.php' > no-more-php-files/.gitignore
$ git status --untracked-files

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

    .gitignore
    file-1.txt
    file-2.txt
    file-3.txt
    file-4.txt
    no-more-php-files/.gitignore
    no-more-php-files/test.rb
    sub-folder/demo.txt
    test.php

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

`no-more-php-files/` dizini altındaki `demo.php` izole oldu. Sağlamasını yapalım:

```
$ git ls-files -o -i --exclude-standard

file.xyz
no-more-php-files/demo.php
sub-folder/demo.xyz
```

Öncelik Sırası

Öncelik önemlidir. Her şey öncelik sırasına göre işlenir.

1. `core.excludesfile` konfigürasyonunda tanımlı olanlar
2. Repo'nun root'undaki `.gitignore`
3. Alt dizinlerdeki `.gitignore`'lar

Diyelim ki `log/` dizini var. Tüm log üreten servisler buraya yazıyor. Nginx buraya `access.nginx` şeklinde yazıyor, Gunicorn ise `gunicorn.log` şeklinde yazıyor. Biz, sadece `*.log` yaparak gunicorn'u izole edebiliriz:

```
$ mkdir log/ && echo '*.log' > log/.gitignore
$ touch log/gunicorn.log
$ git ls-files -o -i --exclude-standard

file.xyz
log/gunicorn.log
no-more-php-files/demo.php
sub-folder/demo.xyz
```

Peki, bir durum oldu, başka bir servis daha kullanmaya başladık ve o servis de `foo.log` şeklinde log üretiyor ve senaryo bu ya, bunu konfigüre edemiyoruz. Hatta bu dosyayı da takip etmek istiyoruz yani izole etmek **istemiyoruz!** Yapmamız gereken `log/.gitignore` dosyasına bir satır daha eklemek:

```
$ echo '!foo.log' >> log/.gitignore
$ touch log/foo.log
$ git status --untracked-files

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

    .gitignore
    file-1.txt
    file-2.txt
    file-3.txt
    file-4.txt
    log/.gitignore
    log/foo.log
    no-more-php-files/.gitignore
    no-more-php-files/test.rb
    sub-folder/demo.txt
    test.php

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

Gördüğünüz gibi `log/foo.log` artık track edilmeye hazır durumda. `log/.gitignore` dosyasındaki sıralama çok önemli:

```
*.log
!foo.log
```

İlk satırda GIT'e `*.log` dosyalarını görme diyoruz. Hemen ikinci satırda da `foo.log` dosyasını boş geçme, takip et diyoruz. `!` tersi anlamında. GIT ilk olarak tüm `*.log` dosyalarını ignore ediyor ve sonra gelen direktife göre `foo.log` dosyasını etmiyor.

Eğer bu ters yazılsaydı yani;

```
!foo.log
*.log
```

olsaydı, GIT önce `foo.log` dosyasını tutacak, sonra gelen satır **TÜM *.log dosyalarını izole et** emrini vereceği için `foo.log` da kaynayıp gidecekti.

Boş Dizinler

Repo'da, içinde dosya olmayan dizinler otomatik olarak görmezden gelinir. `images/` diye bir dizinimiz olsa ve `.gitignore` 'da:

```
images/*.jpg
```

yazsa, `images/` altında sadece `*.jpg` dosyaları olsa, bu dizin komple ignore edilir.

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

    .gitignore

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

Eğer içinde başka bir dosya olursa, mesela `images/test.png` olsa;

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

    .gitignore
    images/test.png

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

durum değişir ve bu dizin artık izole edilmez. Bu dizin ve içinde `*.jpg` olmayan dosyalar repo'nun parçası olur.

Ruby on Rails dünyasından gördüğüm, hoşuma giden bir taktiği paylaşmak istiyorum. Bazı durumlarda dizini korumak gerekir, yani repo'nun bir parçası olması gerekir ama dizin altındaki belli dosyalar ya da tüm dosyalar **ignore** edilmiş olabilir.

Bu durumda `!.gitkeep` devreye girer. Hemen örnek `.gitignore` dosyasına bakalım:

```
/.env
/bin/
*.sqlite3
/project/media/*
/project/fixtures/*
/project/config/settings/development.py
!.gitkeep
```

Sıralama önemli dedik. `!.gitkeep` mutlaka ignore edilenlerden sonra gelmeli. Bu sayede, korumak istediğim herhangi bir dizin altına `.gitkeep` dosyası koyduğum an o dizin track edilmeye başlayacak ve ignore edilmesi gereken dosyalar da ignore edilecektir.

Dosya Silmek, Dosya Adını Değiştirmek

`git mv`

Linus Torvalds, Linux'daki dosya operasyonlarının pek çoğunu GIT'e entegre etmiş. Nasıl ki Unix/Linux'da bir dosyanın adını değiştirmek için;

```
$ mv dosya dosya_yeni_isim
```

kullanılıyorsa, aynı işi;

```
$ git mv dosya dosya_yeni_isim
```

şeklinde yapmak mümkün. Eğer adını ya da yerini değiştireceğimiz dosya revizyon kontrolü altında değilse GIT size uyarı verir:

```
$ git mv file1 file_new_1

fatal: not under version control, source=file1, destination=file_new_1
```

Dosya track ediliyor, unuttunuz ve `mv` işlemini GIT üzerinden değil de, işletim sistemi üzerinden yaptınız.

```
$ mv app.js application.js
$ 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:    app.js

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

    application.js

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

GIT, `app.js` dosyasının silindiğini ve `application.js` dosyasının takip altında olmayan yepyeni bir dosya olduğunu düşündü. Hatta artık `app.js` dosyasını takipten çıkarmamız için bize;

```
(use "git add/rm <file>..." to update what will be committed)
```

bile dedi... Şimdi biz bu değişikliği kayıt altına alalım, yani stage edelim:

```
$ git add .
$ git status

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

    renamed:    app.js -> application.js
```

Ve GIT `app.js` dosyasını silmediğimizi sadece adını değiştirdiğimizi anladı.

```
renamed:    app.js -> application.js
```

GIT, default olarak **Similarity Index** diye bir değere bakar. Eğer bu karşılaştırma minimum **0.5** olması durumunda GIT bu iki dosyanın aynı olduğuna karar verir.

GIT için önemli olan dosya adına değildir, **dosyanın içeriğidir**.

`git rm`

Aynı `mv` gibi, işletim sisteminin bir kopya operasyonu da buradadır. Normalde dosya silmek için;

```
$ rm file
$ rm -r folder/
```

yaparız. Aynı mantıkta önüne bir tek `git` takıyoruz:

```
$ git rm file
$ git rm -r folder
```

Eğer silme işini GIT üzerinden yapmazsak;

```
$ rm index-test.html
$ 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:    index-test.html

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

Silinme durumunu GIT anladı ama staging'e atmadı. Bu işlemi bizim yapmamızı istiyor. Eğer bunu `git rm` ile yapsaydık;

```
$ git rm index-test.html
$ git status

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

    deleted:    index-test.html
```

otomatik olarak staging'e alındı. Tek yapmamız gereken şey commit etmek.

Silme ya da isim değiştirme işlerini kullandığınız **IDE** yerine mutlaka komut satırından yapmanız sizin için daha kolay GIT kullanımı pratiği yapmanızı sağlar.

Bölüm 02

Bu bölümde işleyeceğimiz konular:

- Branch'lerle Çalışmak
- Branch'leri Birleştirmek
- Branch'lerin Çakışması: Conflict
- Branch'leri Birleştirmek: `rebasing`
- Branch Rebase Sırasında Çakışma: **Rebase Conflict**
- Değişiklikleri Görüntülemek: `git diff`
- Etiketlemek Nedir?: `git tag`

Branch’lerle Çalışmak

GIT’in bence en önemli özelliği branch mekanizmasıdır. O an için çalıştığınız yani içinde bulunduğunuz branch her ne ise, o branch’den ya da başka bir branch’den **N tane**, belkide **onlarca**, **yüzlerce** kopya çıkartabilirsiniz ve bu işlem **SIFIR MALİYETLİ** bir işlemdir.

GIT’in olmadığı ya da herhangi bir revizyon kontrol sisteminin olmadığı bir ortamda çalışıyorsunuz ve projenin belli bir aşamasında, bir özelliği denemek istiyorsunuz. Ne yaparsınız? Ya projeyi ya da o çalıştığınız dosyayı kopyalarsınız ve o kopya üzerinde çalışırsınız.

Düşünsenize, 500 Megabytle’ık bir proje olsa, ufacık bir deneme için, ne olur ne olmaz işi garantiye almak için, komple kopya aldınız. Diskinizden **500MB** daha yer işgal ettiniz. Böyle 5-10 kere kopya yapsanız **Gigabyte**’lar seviyesinde disk alanı harcamış olursunuz.

Diyebilirsiniz ki;

> Amaaaan, ne olacak? bende Terrabyte seviyesinde disk var, 3-5 Gigabyte’ın lafı mı olur?

Hadi diyelim yer sorunu sıkıntı değil. 20 tane kopya yaptınız;

```
proje        # bu esas dizin!
proje_yedek
proje_yedek_1
proje_yedek_2
proje_yedek_esas_1
proje_yedek_esas_1_deneme
proje_yedek_esas_1_deneme_son
proje_yedek_3
proje_yedek_3_son
:
:
```

gibi böyle çılgıncasına dizinler var bilgisayarınızda. Peki neler değişti? Değişen dosyalar hangileri? Bunların listesini tuttunuz mu? Son aşamadan `proje/` dizinini altına hangi dosyaları atacaksınız?

İşte bu ve buna benzer durumlar GIT tarafından süper basit bir şekilde çözümleniyor:

```
$ git branch # repo’daki branch’leri listele
* master
```

GIT bize çalıştığımız kopyayı, branch'i söylüyor: **master branch**. Haydi deneme yapalım, aklımıza bir şey geldi:

```
$ git branch idea # adı idea olan yeni bir branch oluştur
$ git branch
  idea
* master
```

İçinde bulunduğumuz branch `*` ile işaretli. Şimdi yeni oluşturduğumuz **idea** branch'ine geçelim:

```
$ git checkout idea
Switched to branch 'idea'
```

Artık bu alanda istediğimiz her şeyi deneyebiliriz. Birkaç yeni dosya ekleyelim:

```
$ touch testing-file-ab-{1,2}.sh
$ git add .
$ git commit -m 'added 2 test files'
$ ls

file-1.txt
file-2.txt
file-3.txt
file-4.txt
file.xyz
log
no-more-php-files
sub-folder
test.php
testing-file-ab-1.sh
testing-file-ab-2.sh

$ git log --graph --decorate --oneline --all

* b34155b6b819 (HEAD -> idea) added 2 test files
* 1304ac22cd97 (master) Example commit
* 258f67c2e2cd [root] Initial commit
```

Bu çıktı bize şunu ifade ediyor:

1. Toplam 3 tane commit var
2. Çalıştığım yer **idea** branch'i (*HEAD*)

Şimdi master'a dönüp orada directoy list alalım:

```
$ git checkout master
Switched to branch 'master'

$ ls

file-1.txt
file-2.txt
file-3.txt
file-4.txt
file.xyz
log
no-more-php-files
sub-folder
test.php
```

İki branch arasındaki fark ne? master branch'de `testing-file-ab-1.sh` ve `testing-file-ab-2.sh` dosyaları yok. Hatta şöyle sağlamasını yapalım:

```
$ git diff HEAD idea --name-only # git help diff

testing-file-ab-1.sh
testing-file-ab-2.sh
```

idea branch'inde denemelerimizi tamamladık ve bu iki dosyayı artık **master** branch'e almaya karar verdik. Bunu yapmak için yöntemlerden bir tanesi branch'leri birleştirmek yani **merge** etmek!

Branch'leri Birleştirmek: `git merge`

Önce değişiklikleri eklemek istediğimiz yani güncelleme yapmak istediğimiz branch'e geçiyoruz. Sonra, içinde bulunduğumuz brach'e birleştirmek istediğimiz branch'i `merge` komutu ile parametre olarak geçiyoruz.

Örnek repoda, **idea** branch'indeki değişiklikleri **master** branch'e aktaracağız:

Strateji: Fast-Forward

```
$ git log --graph --decorate --oneline --all

* 2546539c16e9 (idea) testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 (HEAD -> master) Example commit
* 258f67c2e2cd [root] Initial commit

$ git checkout master # güncelleyeceğimiz branch'e geçtik
$ git merge idea      # idea branch'indeki değişiklikleri master'a al

Updating 1304ac22cd97..2546539c16e9
Fast-forward
 testing-file-ab-1.sh | 0
 testing-file-ab-2.sh | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testing-file-ab-1.sh
 create mode 100644 testing-file-ab-2.sh
```

Şimdi ne oldu ? Satır satır bakalım:

```
Updating 1304ac22cd97..2546539c16e9
```

`1304ac22cd97` bulunduğumuz yerdi. `2546539c16e9` ise **idea** branch'inin HEAD'iydi.

```
Fast-forward
```

Kaset / teyp çalarları hatırlayan varsa, kaset-çalar'ın üzerinde `PLAY`, `STOP`, `RECORD`, `PAUSE`, `REW`, `FF` gibi düğmeler vardı... Kaseti ileri sarmak için bu `FF` yani **Fast Forward** düğmesine basardık.

İşte GIT'de aynen bu kaset kayıt cihazı gibi ileri sarma işi yaptı. Log'a bakıldığında, linear (*doğrusal*) bir durum görünüyor. **idea** branch'inde değişiklik olmasına rağmen master olduğu yerde kalmış. Yani master branch'de hiçbir şey değişmediği için GIT:

> Hmmm... Bu merge işini yaparken sadece pointer'ın yerini değişerek operasyonu tamamlayabilirim...

diye düşünüp birleştirme işlemini **Fast Forward stratejisi** kullanarak yaptı. Gayet temiz bir operasyon oldu.

```
2 files changed, 0 insertions(+), 0 deletions(-)
```

Zaten hangi dosyalar olduğunu da biliyorduk. Aynen tahmin ettiğimiz ve istediğimiz gibi oldu her şey. Son durum nedir?

```
$ git log --graph --decorate --oneline --all

* 2546539c16e9 (HEAD -> master, idea) testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 Example commit
* 258f67c2e2cd [root] Initial commit
```

Hem **master** hem de **idea** branch'i aynı noktayı işaret ediyorlar. Önerilen, benim de hep yaptığım şey şu; branch'i merge ettikten sonra, artık o branch'le işimin kalmadığını düşünüyorum ve siliyorum:

```
$ git branch -d idea
Deleted branch idea (was 2546539c16e9).

$ git branch -v
* master 2546539c16e9 testing commit
```

Strateji: Recursive

Şimdi master branch'den yeni bir branch oluşturalım:

```
$ git checkout -b development           # branch'i oluşturup otomatik olarak checkout
yaptık
Switched to a new branch 'development'
```

Bir tane yeni dosya ekleyelim:

```
$ touch global.js && git add global.js
$ git commit -m 'main js file added'

[development 7e3c5612484b] main js file added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 global.js

$ git log --graph --decorate --oneline --all

* 7e3c5612484b (HEAD -> development) main js file added
* 2546539c16e9 (master) testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 Example commit
* 258f67c2e2cd [root] Initial commit
```

Bu sırada aklımıza bir şey geldi, hemen master branch'e dönüp o unuttuğumuz işi yapalım:

```
$ git checkout master
Switched to branch 'master'

$ touch server-{1,2}.rb && git add server-{1,2}.rb
$ git commit -m 'server files added'

[master e2fba879b0df] server files added
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 server-1.rb
 create mode 100644 server-2.rb

 $ git log --graph --decorate --oneline --all

 * e2fba879b0df (HEAD -> master) server files added
 | * 7e3c5612484b (development) main js file added
 |/
 * 2546539c16e9 testing commit
 * b34155b6b819 added 2 test files
 * 1304ac22cd97 Example commit
 * 258f67c2e2cd [root] Initial commit
```

İlk kez böyle dallı budaklı bir log ile karşı karşıyayız. `2546539c16e9` numaralı commit'ten iki tane türeme olmuş. Bunlar;

1. `7e3c5612484b (development) main js file added`
2. `e2fba879b0df (HEAD -> master) server files added`

Yani `7e3c5612484b` ve `e2fba879b0df` komitlerinin atası: `2546539c16e9`

```
$ git show --pretty=%h 7e3c5612484b^ # git help show
2546539c16e9

$ git show --pretty=%h e2fba879b0df^
2546539c16e9
```

1. master branch'deki 2 dosya: `server-1.rb` ve `server-2.rb` development branch'inde yok
2. development branch'indeki `global.js` de master branch'de yok

Biz ne yapmak istiyoruz ? development branch'deki değişiklikleri master branch'e aktarmak... O zaman;

```
$ git checkout master # master branch'e geçtik
Switched to branch 'master'

$ git merge development
```

İşte bu noktada karşımıza pat diye mesaj editörü çıktı ve ne görüyoruz?

```
Merge branch 'development'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

Otomatik üretilmiş bir commit mesajından başka bir şey değil bu! GIT bize;

> Lütfen bu birleştirmeyi açıklayan bir mesaj yaz!

diye uyarıda bulunuyor. Hiçbir şey yazmadan kaydedip çıkıyorum. Mini not: Eğer `$EDITOR` environment variable ataması yapmadıysanız ya da Temel Konfigürasyon Öğeleri - core.editor ayarı yapmadıysanız karşınıza `vi` text editörü çıkacak. Bu durumda mesajı kaydedip çıkmak için: `:wq` yapmanız gerekecek.

```
Merge made by the 'recursive' strategy.
 global.js | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 global.js

 $ git log --graph --decorate --oneline --all

 *   5424468beb69 (HEAD -> master) Merge branch 'development'
 |\
 | * 7e3c5612484b (development) main js file added
 * | e2fba879b0df server files added
 |/
 * 2546539c16e9 testing commit
 * b34155b6b819 added 2 test files
 * 1304ac22cd97 Example commit
 * 258f67c2e2cd [root] Initial commit
```

İşte **Recursive strateji** ile yapılan merge sonucunda nur topu gibi bir **merge commit**'imiz oldu. Buna **merge bubble** da denir.

```
5424468beb69 (HEAD -> master) Merge branch 'development'
```

Recursive olmasının ne sıkıntısı ya da faydası var ? Şöyle bakalım. Ne demiştim? İşimiz bitince branch'i siliyoruz. Yani şimdi development branch'i uçurup log'a tekrar bakalım:

```
$ git branch -d development
Deleted branch development (was 7e3c5612484b).

$ git log --graph --decorate --oneline --all

*   5424468beb69 (HEAD -> master) Merge branch 'development'
|\
| * 7e3c5612484b main js file added
* | e2fba879b0df server files added
|/
* 2546539c16e9 testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 Example commit
* 258f67c2e2cd [root] Initial commit
```

Bu log'a baktığımızda, ilk anda `5424468beb69` numaralı commit'in bir merge commit olduğunu, hatta bunun development branch'inden türediğini görebiliyoruz. Bu kimileri için faydalı olabilir. Ben **linear history** yani doğrusal, satır satır log görmeyi sevenlerdenim.

Peki hemen şu soru gelmeli:

> Peki benzer bir durumda, merge commit olmadan, linear history yapmak mümkün mü?

Aynı örnekteki gibi, master ve development branch'lerinde farklılıklar olacak ve biz merge commit olmadan merge yapacağız? Evet mümkün... Buna **rebase** işlemi deniliyor. Yaptığımız şey de **branch rebasing** oluyor. Bu konuya geleceğiz.

GIT default olarak branch'leri merge ederken **Fast-Forward** stratejisini deniyor ilk olarak. Eğer istersek bunu değiştirebiliriz. İster konfigürasyon seviyesinde ister işlem seviyesinde.

Eğer default olarak branch'lerin merge işlemini Recursive strateji kullanarak yapmasını isterseniz:

```
$ git config --global merge.ff false
```

eğer işlem seviyesinde yapmasını isterseniz de:

```
$ git checkout -b no-ff # demo amaçlı no-ff adında bir branch aç ve checkout yap
Switched to a new branch 'no-ff'

$ touch file-for-no-ff.txt && git add file-for-no-ff.txt
$ git commit -m 'no-ff commit'
[no-ff 8810d33ff41f] no-ff commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file-for-no-ff.txt

$ git checkout master
$ git merge no-ff --no-ff
Merge made by the 'recursive' strategy.
 file-for-no-ff.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file-for-no-ff.txt

$ git log --graph --decorate --oneline --all

*   c83d33173bbe (HEAD -> master) Merge branch 'no-ff'
|\
| * 8810d33ff41f (no-ff) no-ff commit
|/
*   5424468beb69 Merge branch 'development'
|\
| * 7e3c5612484b main js file added
* | e2fba879b0df server files added
|/
* 2546539c16e9 testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 Example commit
* 258f67c2e2cd [root] Initial commit
```

`--no-ff` ile otomatik olarak birleştirme sonunda merge commit üretebilirsiniz. Hatta, önce ne olduğuna bakıp sonrasında kendiniz commit'i manual olarak yapabilirsiniz:

```
$ git checkout -b no-ff-no-commit
$ touch file-for-no-ff-no-commit.txt && git add $_
$ git commit -m 'commit for no-ff and no-commit demo'
[no-ff-no-commit 4d32113f1100] commit for no-ff and no-commit demo
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file-for-no-ff-no-commit.txt

$ git checkout master
$ git merge no-ff-no-commit --no-ff --no-commit
Automatic merge went well; stopped before committing as requested
```

`--no-commit` yüzünden merge işlemi tamamlandı ama merge commit yapılmadı. Bunu biz yapacağız. Hatta bu noktada eğer gerekiyorsa başka dosyalar eklemek çıkartmak da mümkün:

```
$ git status

On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    new file:   file-for-no-ff-no-commit.txt
```

Haydi bitirelim bu işlemi:

```
$ git commit -m 'this is manual merge commit. merged with branch: no-ff-no-commit'
[master dc0c3779544f] this is manual merge commit. merged with branch: no-ff-no-commit

$ git log --graph --decorate --oneline --all

*   dc0c3779544f (HEAD -> master) this is manual merge commit. merged with branch: no-
ff-no-commit
|\
| * 4d32113f1100 (no-ff-no-commit) commit for no-ff and no-commit demo
|/
*   c83d33173bbe Merge branch 'no-ff'
|\
| * 8810d33ff41f (no-ff) no-ff commit
|/
*   5424468beb69 Merge branch 'development'
|\
| * 7e3c5612484b main js file added
* | e2fba879b0df server files added
|/
* 2546539c16e9 testing commit
* b34155b6b819 added 2 test files
* 1304ac22cd97 Example commit
* 258f67c2e2cd [root] Initial commit
```

ASCII'den grafik çizer gibi... Dallar budaklar hepsi burada :)

Teyp Fotoğrafı © vicent.zp

Branch’lerin Çakışması: Conflict

Aynı dosyalar üzerinde çalışınca **conflict** yanı çakışma yaşamak kaçınılmazdır. Duruma göre bu çakışmalar çok can sıkıcı olabilir. Bazen kabus haline bile dönüşebilir. Halk arasında **conflict yemek** olarak da bilinir.

GIT sağ olsun çözüm yöntemlerini öğrenip (*daha doğrusu kaydedip*) benzer çakışma olduğu zaman otomatik olarak çözebiliyor. Bunun için konfigürasyon ayarı yapmak gerekiyor:

```
git config --global rerere.enabled true
```

rerere yi ilk gördüğümde bir Beşiktaş taraftarı olarak aklıma Galatasaray’ın **rerere rarara** tezahüratı gelmişti :) Ne demek rerere ? **Reuse Recorded Resolution** (*of conflicted merges*) yani: branch’leri merge ederken (*birleştirirken*) kaydedilen çözüm şeklini, aynı çakışma olduğu an tekrar kullan.

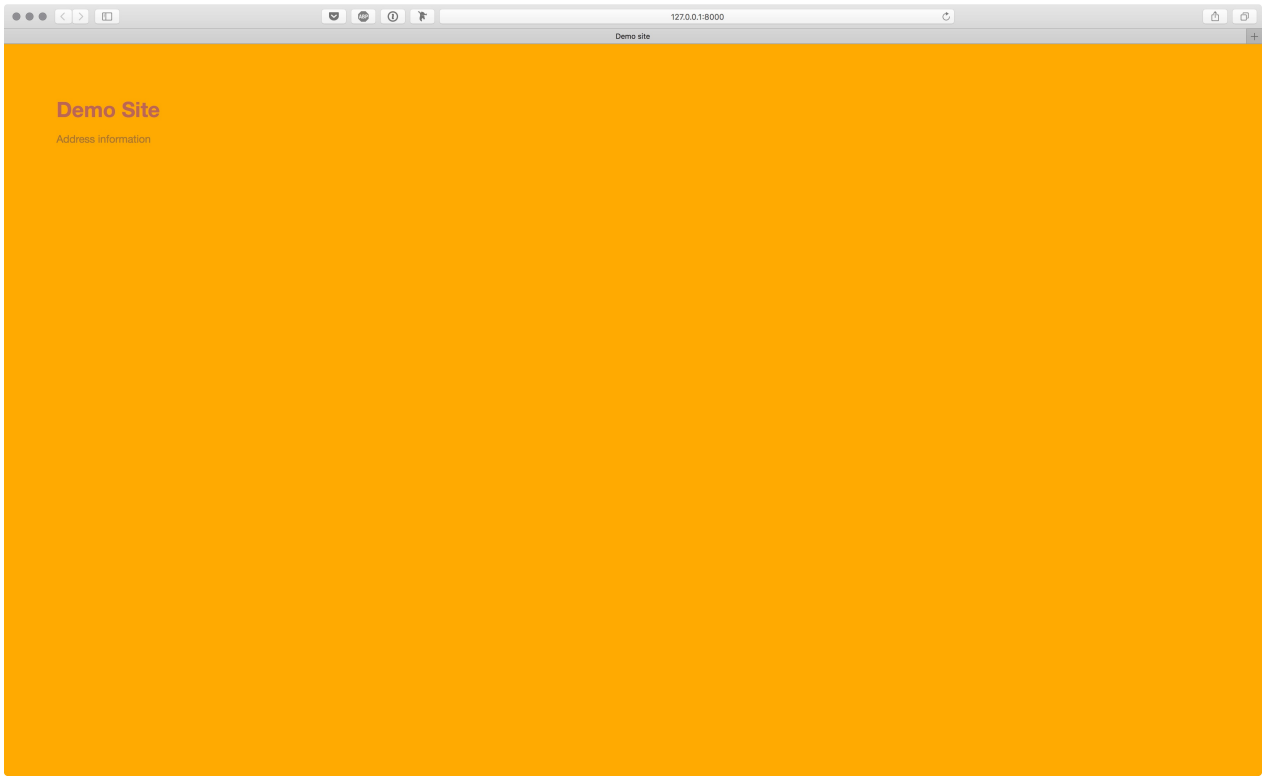
Bir web uygulaması yapıyoruz, yeni bir feature geliştirmek istiyoruz. Uygulama yapısı aşağıdaki gibi:

```
.
├── public
│   ├── css
│   │   └── application.css
│   ├── images
│   │   └── photo.jpg
│   └── js
│       └── global.js
└── index.html
```

Yeni feature için bir branch açıp çalışalım:

```
$ git checkout -b feature
Switched to a new branch 'feature'

# index.html’e eklemeler yaptık, css’e ek yaptık
$ git add .
$ git commit -m 'Footer information added'
[feature c5084b90ab6e] Footer information added
 2 files changed, 6 insertions(+)
```




```
$ git log --graph --decorate --oneline --all

* c5084b90ab6e (HEAD -> feature) Footer information added
* 60ee4fd58668 (master) Demo site initiated
* fea8c5d8e385 [root] Initial commit
```

Şimdi tam bu esnada, bir telefon geldi ve hemen **master** branch'deki `index.html` dosyasında bir değişiklik yapmanız gerektiğini öğrendiniz. Henüz geliştirmekte olduğunuz şeyi de bitirmediniz... Ne lazım?

```
$ git checkout master

# gereken değişikliği yaptınız:

$ git diff head:index.html feature:index.html

diff --git a/index.html b/index.html
index d62d871da6ef..4fbafe3dfd2b 100644
--- a/index.html
+++ b/index.html
@@ -9,8 +9,10 @@
     </head>
     <body>
         <header>
-            <h1>Demo Site 2017</h1>
+            <h1>Demo Site</h1>
         </header>
-        <script src="public/js/global-1.js"></script>
+
+        <footer>Address information</footer>
+        <script src="public/js/global.js"></script>
     </body>
 </html>
```

Şimdi commit edelim:

```
$ git add .
$ git commit -m 'added: 2017 text to heading1 and some other fixes'

[master 8f9bb3474c79] added: 2017 text to heading1 and some other fixes
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Log'da durum nasıl görünüyor ?

```
$ git log --graph --decorate --oneline --all

* 8f9bb3474c79 (HEAD -> master) added: 2017 text to heading1 and some other fixes
| * c5084b90ab6e (feature) Footer information added
|/
* 60ee4fd58668 Demo site initiated
* fea8c5d8e385 [root] Initial commit
```

`60ee4fd58668` ID'li commit'den türeyen 2 commit var:

1. `c5084b90ab6e`
2. `8f9bb3474c79`

Bu iki commit arasındaki fark ne?

```
$ git diff c5084b90ab6e 8f9bb3474c79 # feature ile HEAD arasındaki diff bu yani
$ git diff feature HEAD              # aynı şey

diff --git a/index.html b/index.html
index 4fbafe3dfd2b..d62d871da6ef 100644
--- a/index.html
+++ b/index.html
@@ -9,10 +9,8 @@
     </head>
     <body>
         <header>
-            <h1>Demo Site</h1>
+            <h1>Demo Site 2017</h1>
         </header>
-
-        <footer>Address information</footer>
-        <script src="public/js/global.js"></script>
+        <script src="public/js/global-1.js"></script>
     </body>
 </html>
diff --git a/public/css/application.css b/public/css/application.css
index e166ec262bd6..018697659236 100644
--- a/public/css/application.css
+++ b/public/css/application.css
@@ -9,8 +9,4 @@ h1, h2, h3, h4, h5, h6 {
     color: #b65;
     padding: 0;
     margin: 1rem 0;
-}
-
-footer {
-    color: #a73;
 }
\ No newline at end of file
```

Birinde olan diğerinde yok ya da diğerinde tamamen başka bir şey var. Biz master branch'deyiz ve `git merge feature` dediğimiz an conflict yiyeceğimiz ap açık ortada. Karşımızdaki seçenekler:

1. Kodu yazan biz olduğumuz için ne olması gerektiğini biliyoruz, elle gereken düzeltmeleri yapacağız.
2. Doğru olan kısım **master** branch'deki kısım, esas olan o!
3. Doğru olan kısım **feature** branch'deki kısım, esas olan o!
4. Amaaaan, bana ne ya, vazgeçiyorum! kim merge ederse etsin! :)

Haydi o zaman:

```
$ git merge feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Recorded preimage for 'index.html'
Automatic merge failed; fix conflicts and then commit the result.
```

Evet, GIT, otomatik olarak `index.html` 'i **merge** etmek istedi ama bu dosyada **merge conflict** ile karşılaştı ve olası çözüm için kayda geçti:

```
Recorded preimage for 'index.html'
```

Durum ne?

```
$ git status

On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:

    modified:   public/css/application.css

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

    both modified:   index.html
```

`public/css/application.css` bunda sıkıntı yok. `both modified` ne demek? Kelime anlamı olarak **her ikisi de değişti** gibi bir şey çıkıyor. Burada bahsi geçenler kim? Az önce karşımızdaki seçeneklerden bahsetmiştim. Doğru olan kısım **master** ya da **feature**...

GIT, birleştirmek istediğimiz branch'i **ONLARIN** yani **theirs**, içinde bulunduğumuz branch'i de **BİZİM** yani **ours** olarak sınıflandırıyor. Bir şeyin `both modified` olması demek, hem bizim tarafın, hem de onların tarafın modifiye olması demek.

Şimdi son seçenek olan: **Bana ne! vazgeçtim** i yapalım. master branch için (*şu an master branch'deyiz ya*) merge'den önceki son commit hangisiydi?

```
$ git reset --hard 8f9bb3474c79
HEAD is now at 8f9bb3474c79 added: 2017 text to heading1 and some other fixes
```

Her commit, aslında zamanın fotoğrafını çekmek değil miydi? Şimdi o an'a geri döndük. Özellikle `--hard` kullandık ki geride bir şey kalmasın. `git reset` konusunu ileride göreceğiz. Şimdi tekrar merge edip conflict'e geri dönelim:

```
$ git merge feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Recorded preimage for 'index.html'
Automatic merge failed; fix conflicts and then commit the result.
```

Şu sorunlu `index.html`'e bir bakalım:

```
<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Demo site</title>
        <link rel="stylesheet" href="public/css/application.css" type="text/css" />
    </head>
    <body>
        <header>
            <h1>Demo Site 2017</h1>
        </header>
<<<<<<< HEAD
        <script src="public/js/global-1.js"></script>
=======

        <footer>Address information</footer>
        <script src="public/js/global.js"></script>
>>>>>>> feature
    </body>
</html>
```

Bu dosyada dikkat edeceğimiz şeylerin başında gelen işaretle:

1. `<<<<<<<`
2. `>>>>>>>`
3. `=======`

`<<<<<<< HEAD` diye ifade işaretlenen yer **master** branch'de olan fark. Yani **BİZİM** yani `--ours`. `>>>>>>> feature` diye işaretlenen yer **feature** branch'de olan fark. Yani **ONLARIN** yani `--theirs`. `=======` ise ayraç. Üst ile alt kısmı ayırıyor.

Şimdi;

1. Ya doğru olanı, olması gerekeni biz bildiğimiz için kendi kafamıza göre gereken

değişikliği yapacağız
2. Ya doğru olan bizim taraf, onların tarafı unut diyeceğiz
3. Ya da doğrusu onların, bizimkini çöpe at diyeceğiz

Bu örnek süper basit ve çok rahat anlaşılıyor ne olduğu. Daha karmaşık kompleks projelerde bu işi çözmek için 3.parti araçlar kullanmak bile gerekebiliyor.

Ben "esas olan onların yaptıklarıdır, ben zaten uydurdum" diyorum ve;

```
$ git checkout --theirs index.html
# yok bizimkini al: git checkout --ours index.html
$ cat index.html
```

ve;

```
<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Demo site</title>
        <link rel="stylesheet" href="public/css/application.css" type="text/css" />
    </head>
    <body>
        <header>
            <h1>Demo Site</h1>
        </header>

        <footer>Address information</footer>
        <script src="public/js/global.js"></script>
    </body>
</html>
```

`checkout` sadece branch'ler arası geçiş dışında da başka fonksiyonları olan bir komut. `git reset` konusunda değineceğim ama hızlıca; `git checkout -- DOSYA` aslında `git status` komutundan aşina olduğumuz bir şey:

```
Changes not staged for commit:
  (use "git checkout -- <file>..." to discard changes in working directory)
```

derki: *to discard changes in working directory* yani, çalıştığınız yerde/dizinde/branch'de, yaptığınız değişiklikleri çöpe atmak için: `git checkout -- <file>...` kullanın!

Bu mini nottan sonra bari en azından **2017**'yi ekleyeyim:

```
$ git add index.html
$ git status

On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
    modified:   public/css/application.css
```

Artık commit yapacak kıvama geldi:

```
$ git commit -m 'fixed: funny merge conflict'
Recorded resolution for 'index.html'.
[master d1d1d648b034] fixed: funny merge conflict
```

Dikkat ettiyseniz GIT hemen atladı ve bu çözümü öğrendiğini söyledi hemen. Log ne alemde?

```
$ git log --graph --decorate --oneline --all

*   d1d1d648b034 (HEAD -> master) fixed: funny merge conflict
|\
| * c5084b90ab6e (feature) Footer information added
* | 8f9bb3474c79 added: 2017 text to heading1 and some other fixes
|/
* 60ee4fd58668 Demo site initiated
* fea8c5d8e385 [root] Initial commit
```

Nur topu gibi bir **merge bubble/commit** ile işimize devam ediyoruz. Hemen master'daki güncellemeleri feature'a da alalım, işimiz henüz bitmedi:

```
$ git checkout feature
Switched to branch 'feature'

$ git merge master
Updating c5084b90ab6e..d1d1d648b034
Fast-forward
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Temiz bir **Fast-forward** oldu. Log ne durumda?

```
$ git log --graph --decorate --oneline --all

*   d1d1d648b034 (HEAD -> feature, master) fixed: funny merge conflict
|\
| * c5084b90ab6e Footer information added
* | 8f9bb3474c79 added: 2017 text to heading1 and some other fixes
|/
* 60ee4fd58668 Demo site initiated
* fea8c5d8e385 [root] Initial commit
```

Bulunduğumuz yer yani `HEAD`, hem **master**'ı hem de **feature**'ı işaret ediyor. History'nin (*yani log'un*) temiz olması için bize ne lazım ?

Branch’leri Birleştirmek: `git rebase`

Branch Rebase Sırasında Çakışma: Rebase Conflict

Değişiklikleri Görüntülemek: `git diff`

@wip

`git whatchanged`

@wip

Etiketlemek Nedir?: `git tag`

Bölüm 03

Bu bölümde işleyeceğimiz konular:

- Commit'leri Birleştirmek: **Interactive Rebasing**
- Commit'leri Bölmek
- Cımbızla Commit'i Almak: **Cherry Picking**
- Hataları İşlemleri Geri Almak ya da Vazgeçmek: `reset revert amend`
- Commit'e Not Eklemek
- Her şey Kayıt Altında! En az 90 gün: `git reflog`

Commit’leri Birleştirmek: Interactive Rebasing

Commit’leri Bölmek

@wip

Cımbızla Commit’i Almak: Cherry Picking

Hataları İşlemleri Geri Almak ya da Vazgeçmek

`git reset`

@wip

`--amend`

@wip

`git revert`

@wip

Anlık Değişiklikleri Saklamak: `git stash`

@wip

Commit’e Not Eklemek

Her şey Kayıt Altında! En az 90 gün: `git reflog`

Bölüm 04

Bu bölümde işleyeceğimiz konular:

- Remote Kavramı Nedir? Remote'larla Çalışmak
- GitHub, BitBucket ve GitLab ile Çalışmak
- Kendi GIT Reponuzu Yapın!
- Repo içinde Repo: `git submodule`

Remote Kavramı Nedir? Remote'larla Çalışmak

@wip

Tracking Branch / Upstream

@wip

`git fetch` ile `git pull` Farkı ?

@wip

`git pull --rebase` ile `git pull` Farkı ?

@wip

Kendi GIT Reponuzu Yapın!

@wip

GitHub, BitBucket ve GitLab ile Çalışmak

Repo içinde Repo: `git submodule`

Bölüm 05

Bu bölümde işleyeceğimiz konular:

- Commit Öncesi ya da Sonrası Otomasyonu: **GIT Hook'ları**
- Bundle Nedir?
- Commit'inizi İmzalayın
- Revizyonları Sorgulamak
- Commit'leri Sorgulamak: `blame`
- Bisect Nedir?
- Yardımcı Araçlar
- Faydalı İpuçları

Commit Öncesi ya da Sonrası Otomasyonu: GIT Hook’ları

@wip

Bundle Nedir?

@wip

Commit’inizi İmzalayın

@wip

Revizyonları Sorgulamak

@wip

Commit’leri Sorgulamak

`git blame`

@wip

Bisect Nedir?

@wip

Yardımcı Araçlar

`tig`

@wip

`git extras`

@wip

Faydalı İpuçları

@wip