# Working With Commits

İyi commit mesajları yazmak, git log/git show ile geçmişte gezinmek, git commit -am, ve git commit --amend'in var olan bir commit'i yerinde değiştirmek yerine YENİ bir commit oluşturduğunu anlamak. HEAD ve HEAD~1/HEAD~2 referansları.

**Başlangıç** 25 dk

Son güncelleme: 27 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/working-with-commits

---

Artık nasıl commit oluşturacağını biliyorsun. Bu ders, commit'lerini gerçekten *faydalı* hale getirmekle ilgili — hem gelecekteki kendine, hem kod tabanının neden şu anki hâline geldiğini anlamaya çalışırken `git log`'u okuyacak her takım arkadaşına. Bir repository'nin geçmişi bir tür dokümantasyondur; "fix" ve "update" adlı commit'lerle dolu bir repository hiçbir şeyi dokümante etmez.

## İyi Commit Mesajları Yazmak

İyi bir commit mesajı tek bir soruyu yanıtlar: bu değişiklik **neden** yapıldı? *Ne* yapıldığı zaten `git diff`'te görünür — mesajın bunu tekrarlamasına gerek yok.

```
Kötü:  fix bug
Kötü:  update UserService.java
İyi:   Fix NullPointerException when user has no roles assigned
İyi:   Add retry logic to handle transient database timeouts
```

Çoğu takımın kullandığı konvansiyon, emir kipiyle yazılmış kısa (~50 karakter) bir özet satırıdır ("Add", "Fix", "Remove" — "Added" ya da "Fixes" değil), isteğe bağlı olarak boş bir satır ve *neden*i tek cümleden fazlası gerektiriyorsa daha uzun bir açıklama takip eder.

Bilinmesi gereken, giderek daha yaygınlaşan bir konvansiyon **Conventional Commits**'tir — özeti bir tip önekiyle başlatmak:

```
feat: add password reset endpoint
fix: correct off-by-one error in pagination
docs: update README setup instructions
refactor: extract validation logic into a separate method
```

Bu, Git'in kendisi tarafından zorunlu tutulmaz — bir takım/araç konvansiyonudur (bazı projeler bunu changelog otomatik üretmek için kullanır) — ama profesyonel Java/Spring Boot takımlarında yeterince yaygındır ki tanıman gerekir.

## git log ve Faydalı log Seçenekleri

`git log`'un, varsayılan çok satırlı çıktıdan gündelik kullanımda çok daha faydalı hale getiren seçenekleri vardır:

```
git log --oneline              # commit başına tek satır -- hızlı genel bakış
git log --oneline -5           # yalnızca son 5 commit
git log --author="Ada"         # yalnızca belirli bir yazarın commit'leri
git log --since="2 days ago"   # yalnızca son zamanlardaki commit'ler
git log -- UserService.java    # yalnızca bu dosyayı değiştiren commit'ler
```

Özellikle `git log --oneline`, bir alışkanlık hâline geldiğinde günde onlarca kez çalıştıracağın bir şeydir — taranması kolay, kompakt bir commit-hash + özet görünümü verir.

## git show

`git log` commit'leri listelerken, `git show` sana belirli BİR commit'in **tam içeriğini** gösterir — mesajını ve tam diff'ini:

```
git show 4f2a1c9        # belirli bir commit'i (kısmi) hash ile göster
git show HEAD           # en son commit'i göster
```

Git'in bir hash'i benzersiz olarak tanımlaması için yalnızca yeterli önek gerekir — normal boyutlu bir projede genellikle 7 karakter yeterlidir.

## git commit -am ile Commit Oluşturmak

Git'in **zaten takip ettiği** dosyalar için, `-am` staging ve commit'i tek adımda birleştirir:

```
git commit -am "Fix null check in UserService"
```

Bu, `git add -u` (takip edilen, değiştirilmiş TÜM dosyaları stage'le) ardından `git commit -m "..."` ile aynıdır. Önemli kelime "takip edilen" — `-am`, yepyeni, takip edilmeyen dosyaları stage'lemez; onlar için hâlâ açık bir `git add` gerekir.

> `-am`, düşündüğün tek dosya değil, değiştirilmiş her takip edilen dosyayı stage'ler. Working tree'nde ilgisiz değişiklikler varsa, `-am` onları aynı commit'e paketler. Odaklı bir commit istediğinde açık `git add <file>`'ı tercih et.

## Son Commit'i Değiştirmek — git commit --amend

Az önce commit attıysan ve hemen bir hata fark ettiysen — mesajda bir yazım hatası, ya da dahil etmeyi unuttuğun bir dosya — `--amend` ayrı bir "oops" commit'i oluşturmadan bunu düzeltmene izin verir:

```
git add ForgottenFile.java     # unuttuğun dosyayı stage'le
git commit --amend             # mesajı isteğe bağlı düzenlemek için editör açar
git commit --amend --no-edit   # aynı mesajı koru, yalnızca stage edilenleri ekle
```

```
#!/bin/sh
# Committing, noticing a forgotten file, and amending instead of creating a
# second "oops" commit.

git commit -m "Add UserService with findById method"
# Output:
# [main 8a1b2c3] Add UserService with findById method
#  1 file changed, 8 insertions(+)

# Oops -- forgot the matching test file.
git add UserServiceTest.java
git commit --amend --no-edit

# Output:
# [main a9f3e21] Add UserService with findById method
#  2 files changed, 24 insertions(+)

# Note the new commit hash (a9f3e21) -- it REPLACED 8a1b2c3, it did not edit
# it in place. `git log --oneline -1` now shows only the new commit.
git log --oneline -1
# Output:
# a9f3e21 Add UserService with findById method
```

**Önemli kavram**: `--amend`, var olan commit'i yerinde DEĞİŞTİRMEZ — Git commit'leri değişmezdir (immutable). Aslında olan şey, Git'in eskisinin yerine geçecek **yepyeni bir commit** oluşturması ve branch'ini yeni olana işaret edecek şekilde taşımasıdır. Eski commit teknik olarak bir süre daha var olmaya devam eder (bu kursta daha sonra ele alınacak `git reflog` ile kurtarılabilir), ama artık branch'inin geçmişinin bir parçası değildir.

Bu ayrım önemlidir çünkü ima ettiği şey şudur: eğer o commit'i zaten **push** ettiysen ve başka biri onu pull ettiyse, amend etmek senin geçmişinle onların geçmişi arasında bir sapma yaratır — branch'in artık o pozisyonda onların sahip olduğu kopyadan *farklı* bir commit'e sahip. Kural tam olarak bu yüzden şudur: **yalnızca hâlâ yerel olan, henüz kimsenin pull etmediği commit'leri amend et.** Paylaşılan geçmişi yeniden yazmanın tam tehlikesini, bu kursta daha sonra `git push --force`'a geldiğimizde ele alacağız.

## HEAD'i Anlamak

**HEAD**, Git'in "şu an checkout edilmiş olan commit" için kullandığı isimdir — sabit bir şey değil, bir işaretçidir (pointer). Çoğu zaman HEAD, mevcut branch'inin ucuna işaret eder ve her commit attığında otomatik olarak ileri hareket eder.

```
git log -1 HEAD    # HEAD'in şu an işaret ettiği commit'i göster
```

HEAD'i Git araçlarında ve hata mesajlarında sürekli göreceksin — her zaman "şu anda, checkout edilmiş geçmişimde, burada" anlamına gelir.

## Commit Referansları — HEAD~1, HEAD~2, vb.

Hash'lerini bilmeden, önceki commit'lere HEAD'e *göreceli* olarak referans verebilirsin:

```
git show HEAD~1     # mevcut commit'ten önceki commit
git show HEAD~2     # mevcut commit'ten iki önceki commit
git log HEAD~3..HEAD  # en son 3 commit
```

`HEAD~1` genellikle `HEAD^` olarak da yazılır (normal, tek-ebeveynli bir commit zinciri için ikisi de aynı anlama gelir — `^` ve `~N` yalnızca birden fazla ebeveyni olan merge commit'ler devreye girdiğinde farklılaşmaya başlar, ki bunu Merge dersinde ele alacağız). Bunları bir sonraki derste, Undoing Changes'te, sürekli kullanacaksın — `git reset` gibi komutlar hedeflerini neredeyse her zaman bu şekilde ifade eder.

## Yaygın Hatalar

- **Zaten push edilmiş ve başka biri tarafından pull edilmiş bir commit'i amend etmek.** Bu, paylaşılan geçmişi yeniden yazar — kimsenin o commit'e sahip olmadığından emin olmadıkça kaçın.
- **`-am` kullanıp yanlışlıkla working tree'de duran ilgisiz değişiklikleri commit etmek.**
- **Commit mesajlarını *ne*yin değiştiğini açıklayacak şekilde yazmak** ("UserService'te 3 satır değişti") *neden* yerine — diff zaten neyin değiştiğini gösterir.

## En İyi Pratikler

- Commit'leri küçük ve odaklı tut — commit başına tek mantıksal değişiklik, ilgisiz düzeltmelerin karışımı değil.
- Özet satırını emir kipinde, ~50 karakterin altında yaz.
- Henüz push etmediğin commit'ler için `git commit --amend`'i rahatça kullan — bu, "typo düzeltme" takip commit'leri biriktirmek yerine geçmişi temiz tutar.

# Özet, Cheat Sheet ve Terimler Sözlüğü

## Özet

- Bir commit mesajı *neden*i açıklamalı, yalnızca *ne*yi değil — diff zaten *ne*yi gösterir.
- `git log --oneline` ve `git show`, geçmişte gezinmek için gündelik araçlarındır.
- `-am`, zaten takip edilen dosyaları tek adımda stage'ler ve commit'ler, ama asla yeni, takip edilmeyen dosyaları stage'lemez.
- `--amend`, son commit'in yerine geçen *yeni* bir commit oluşturur — geçmişi asla yerinde düzenlemez, ve yalnızca kimsenin pull etmediği commit'lerde kullanılmalıdır.
- `HEAD` her zaman o an checkout edilmiş commit'ine işaret eder; `HEAD~1` / `HEAD~2` ona göreceli önceki commit'lere referans verir.

## Cheat Sheet

```
git log --oneline              # kompakt geçmiş
git show <hash>                # bir commit'in tam içeriği
git commit -am "message"       # takip edilen değişiklikleri stage'le + commit et
git commit --amend             # son commit'in yerine geç
git commit --amend --no-edit   # yerine geç, aynı mesajı koru
git show HEAD~1                # mevcut commit'ten önceki commit
```

## Terimler Sözlüğü

- **HEAD** — şu an checkout edilmiş commit'ine işaret eden bir pointer.
- **Amend** — son commit'in yerine, ek değişiklikler ve/veya yeni bir mesaj içeren yeni bir commit'i koymak.
- **Conventional Commits** — commit özetlerini bir tip önekiyle (`feat:`, `fix:`, vb.) başlatma konvansiyonu.