

GIT NEDİR ?

Git, yazılım geliştirme dünyasında kullanılan bir dağıtık versiyon kontrol sistemidir. Git'in temel amacı, bir projenin kaynak kodunda yapılan tüm değişiklikleri kaydetmek ve yönetmektir. Böylece proje üzerinde çalışan kişiler, kodun geçmişteki herhangi bir haline geri dönebilir, yapılan değişikliklerin kim tarafından ve ne zaman yapıldığını görebilir.

Git aynı zamanda takım çalışmasını kolaylaştırır. Birden fazla kişi aynı projede aynı anda çalışabilir, kendi dallarında (branch) geliştirme yapabilir ve işler tamamlandığında bu dalları birleştirebilir (merge). Bu sayede projede düzen bozulmaz ve herkes verimli bir şekilde katkı sağlayabilir. Ayrıca Git dağıtık yapısı sayesinde her geliştiricinin bilgisayarında projenin tam bir kopyası bulunduğu için internet olmasa bile çalışmaya devam edilebilir.

Kısaca, Git projeleri güvenli, düzenli ve işbirliğine uygun şekilde yönetmek için kullanılır.

VERSİYON KONTROL SİSTEMİ NEDİR ?

Versiyon kontrol sistemi, bir dosya ya da proje üzerinde yapılan değişiklikleri kaydeden ve yöneten sistemdir. Dosyalarda yapılan her değişiklik bir sürüm (versiyon) olarak saklanır. Böylece önceki sürümlere geri dönmek, dosyalar arasındaki farkları görmek ve yapılan değişiklikleri takip etmek mümkündür.

Bu sistemler sadece bireysel kullanım için değil, ekip çalışmalarında da çok önemlidir. Örneğin bir yazılım projesinde kimin hangi kodu yazdığı, ne zaman ekleme yaptığı veya hangi hatayı düzelttiği kayıt altında tutulur. Bu sayede ekipler düzenli çalışabilir ve ortaya çıkan sorunlar daha kolay çözülebilir.

DAĞITIK VERSİYON KONTROL SİSTEMİ NEDİR ?

Dağıtık versiyon kontrol sistemi, her geliştiricinin kendi bilgisayarında projenin ve geçmişinin tam bir kopyasını tuttuğu sistemdir. Yani sadece sunucuda değil, geliştiricinin bilgisayarında da tüm proje geçmişi vardır.

Bu yapı sayesinde internet bağlantısı olmadan da commit atılabilir, branch açılabilir veya geçmiş incelenebilir. Uzak sunucu sadece değişiklikleri diğer geliştiricilerle paylaşmak ve senkronize olmak için kullanılır. Eğer uzak sunucu kaybolda bile herhangi bir geliştiricinin bilgisayarındaki kopyadan proje tamamen kurtarılabilir.

En bilinen dağıtık versiyon kontrol sistemi Git'tir. Git, hız, güvenlik ve esneklik sağladığı için günümüzde neredeyse tüm yazılım projelerinde tercih edilmektedir.

SENARYO:

Diyelim ki sen ve üç arkadaşın birlikte bir ödev raporu hazırlıyorsunuz. Başta iş oldukça basit görünüyor: Word dosyasını açıyor, yazıyor ve kaydediyorsunuz. Ama zamanla dosya isimleri karışmaya başlıyor:

“ödev_v1.docx”, “ödev_v2.docx”, “ödev_son.docx”, hatta “ödev_son_son.docx”...

Bir süre sonra hangi dosyanın en güncel olduğu belli olmuyor. İşte burada versiyon kontrol sistemi (VCS) devreye giriyor. VCS sayesinde her değişiklik otomatik olarak kaydediliyor, sürümler tutuluyor ve “Acaba en güncel dosya hangisi?” sorunu ortadan kalkıyor.

Şimdi düşün, öğretmenim sana “Ödevi USB’ye atıp bana getir, ben saklayacağım” dedi. Bu durumda tüm geçmiş USB’de olur. Yani bu, merkezi versiyon kontrol sistemi (CVCS) gibi çalışır. Ancak sorun şu: Eğer USB kaybolursa tüm geçmiş de kaybolur. Ayrıca USB’ye ulaşamayan hiç kimse ödevi güncelleyemez. Yani bu sistem işler ama risklidir.

Bir de daha modern bir yöntem düşünelim: Dağıtık versiyon kontrol sistemi (DVCS). Örneğin Git. Burada senin bilgisayarında da arkadaşlarının bilgisayarında da ödevin tam geçmişiyle birlikte birer kopyası vardır. İnternet olmasa bile herkes kendi bilgisayarında çalışmaya devam edebilir. Arkadaşın kendi kopyasında günceller, sen kendi kopyanda çalışırsın. Sonra internet geldiğinde tüm değişiklikleri birleştirirsiniz. Böylece hiçbir şey kaybolmaz, güvenli bir şekilde herkesin elinde aynı geçmiş bulunur.

Özetle, VCS eski sürümleri saklar ve geri dönmeni sağlar. CVCS her şeyi tek bir merkezde toplar ama bağımlılığı yüksektir, risklidir. DVCS (Git) ise her bilgisayarda tam bir geçmişi sakladığından en güvenli ve esnek yöntemdir.

GRUP İÇİNDE HER BİR PROJE ÜZERİNDE YAPILAN DEĞİŞİKLİKLERİ NASIL ORTAK BİR ŞEKİLDE BİRLEŞTİRİRİZ ?

Bir ödevi sen de arkadaşın da kendi bilgisayarınızda ayrı ayrı düzenlediğinizde aslında iki farklı kopya ortaya çıkar. Senin bilgisayarında senin yaptığın değişiklikler vardır, arkadaşının bilgisayarında ise onun yaptığı değişiklikler bulunur. Doğal olarak bir süre sonra “bu ödevin en güncel hali hangisi?” sorusu ortaya çıkar.

Bu sorunu çözmek için bir ortak buluşma noktası gerekir. Bu buluşma noktası genellikle GitHub, GitLab veya Bitbucket gibi platformlarda bulunan uzak depodur. Bu depo, herkesin kendi yaptığı değişiklikleri götürüp bıraktığı bir merkez gibi düşünülebilir. Yani sen de arkadaşın da kendi değişikliklerinizi önce bu uzak depoya gönderirsiniz. Git, bu depoda değişiklikleri karşılaştırır ve mümkünse otomatik olarak birleştirir.

Eğer sen ve arkadaşın farklı satırlarda çalıştıysanız, Git bunları kolayca birleştirir ve ortaya ortak bir dosya çıkar. Böylece ikinizin yaptığı güncellemeler tek bir yerde toplanır. Herkes uzak depodan bu güncel dosyayı kendi bilgisayarına indirerek aynı sürümü görmüş olur.

Ama eğer aynı satır üzerinde farklı değişiklikler yaptıysanız, Git hangi değişikliğin doğru olduğuna karar veremez. Bu durumda dosyanın içine özel işaretler koyar. Bir tarafta senin yaptığın değişiklik, diğer tarafta arkadaşının yaptığı değişiklik görünür. İşte buna conflict (çatışma) denir. Bu noktada siz birlikte karar vermek zorundasınız: Seninki mi kalsın, onunki mi kalsın, yoksa iki değişiklik birleştirilip yeni bir satır mı yazılsın? Siz doğru çözümü seçip dosyayı düzenledikten sonra ortaya yeni ve ortak bir sürüm çıkar.

Sonuç olarak, süreç şöyle işler:

1. Herkes kendi bilgisayarında istediği değişiklikleri yapar.
2. Bu değişiklikler uzak depoya gönderilir.
3. Git, değişiklikleri birleştirmeye çalışır.
4. Eğer sorun yoksa otomatik birleştirme olur.
5. Eğer aynı satır farklı değiştiyse çatışma çıkar, siz manuel olarak düzeltirsiniz.

6. Ortaya çıkan son sürüm uzak depoda saklanır ve herkes bu sürümü bilgisayarına indirerek kullanır.

GIT KAVRAMLARI

REPOSITORY:

Repository, bir projenin dosyalarının ve bu dosyaların geçmişteki tüm değişikliklerinin saklandığı yerdir. Yani sadece kodların değil, kodların zaman içinde nasıl değiştiğinin de tutulduğu bir depodur. Bilgisayarında kendi repository'ni tutabilirsin (lokal repo) ya da GitHub gibi bir uzak sunucuda saklayabilirsin (remote repo). Repository sayesinde projeni güvenli şekilde saklar, başkalarıyla paylaşır ve istediğin zaman eski sürümlere dönebilirsin.

(Repository'yi bir ödev klasörü gibi düşün. Bu klasörün içinde ödevin her hali ve geçmişte nasıl değiştiği kayıtlıdır. Yani sadece son hali değil, eskiden nasıl görüldüğü de saklanır. Böylece istersen eski sayfalara geri dönebilirsin.)

WORKING DIRECTORY:

Working Directory, repository'de kayıtlı dosyaların bilgisayarındaki çalışma alanıdır. Yani senin üzerinde doğrudan çalıştığın proje klasörüdür. Burada dosyaları açar, değiştirir, yeni dosya eklersin ya da silersin. Bu alanda yaptığın değişiklikler otomatik olarak repository'ye kaydedilmez, sadece senin çalışma alanında görünür.

(Working Directory, ödev klasöründen çıkardığın ve masanın üstünde çalıştığın sayfalardır. Sen bu sayfaların üzerine yazı yazarsın, bilgiyle silersin, yeni sayfa eklersin. Yani aktif olarak üzerinde değişiklik yaptığın yer burasıdır.)

STAGING AREA:

Staging Area, commit atmadan önce hangi değişiklikleri kaydedeceğini seçtiğin ara alandır. Buraya dosya eklediğinde aslında Git'e "bu dosyadaki değişiklikleri bir sonraki commit'e dahil et" demiş olursun. Yani commit'e hazırlık yapılan bölgedir. Staging Area sayesinde sadece istediğin değişiklikleri commit'e ekleyebilirsin, hepsini aynı anda kaydetmek zorunda değilsin.

(Staging Area, ödevini öğretmene teslim etmeden önce hazırladığın dosyalık gibidir. Çalışma masasındaki tüm sayfaları koymazsın, sadece istediğin sayfaları seçip dosyalığa eklersin. Yani commit etmeye (teslim etmeye) hazır hale getirdiğin kısımlar burada durur.)

COMMIT:

Commit, staging area'ya aldığın değişikliklerin repository'ye kalıcı olarak kaydedilmesidir. Her commit bir anlık görüntü (snapshot) gibidir. Hangi dosyaların değiştiği, bu değişiklikleri kimin yaptığı, hangi tarihte yapıldığı ve commit mesajı kayıt altına alınır. Commit sayesinde projenin geçmişine dönebilir, yapılan tüm değişiklikleri takip edebilir ve gelişim sürecini güvenli şekilde saklayabilirsin.

(Commit, dosyalığa koyduğun sayfaları ödev klasörüne (repository'ye) kalıcı olarak koymaktır. Artık bu sayfalar geçmişin bir parçasıdır. Ne zaman teslim ettiğin, hangi değişiklikleri yaptığın ve neden yaptığın kayıt altında olur.)

ÖZET:

- Working Directory → Senin üzerinde çalıştığın proje klasörü.
- Working Directory → Masanın üstünde çalıştığın sayfalar
- Staging Area → Commit öncesi hazırlık yapılan alan.
- Staging Area → Öğretmene vermeden önce dosyalığa koyduğun sayfalar
- Commit → Değişikliklerin kalıcı olarak kaydedilmesi.
- Commit → Dosyalıktaki sayfaların klasöre (geçmişe) kalıcı olarak eklenmesi
- Repository → Projenin ve geçmişin tutulduğu depo.
- Repository → Ödev klasörü (geçmişle birlikte)

Bir de günlük hayat benzetmesiyle:

1. **Working Directory (masa)** → Ödev sayfalarını masaya koyup üzerinde çalışıyorsun.
2. **Staging Area (dosyalık)** → Bitirdiğin ve öğretmene vermeye hazır olan sayfaları dosyalığa koyuyorsun.
3. **Repository (ödev klasörü)** → Dosyalıktaki sayfaları klasöre koyuyorsun, artık geçmişin bir parçası oluyor.
4. **Commit (fotoğraf/iz)** → O anda klasöre koyduklarını tarih, açıklama ve kim tarafından eklendi bilgisiyle kaydediyorsun.

Bu şekilde bakınca Git'in işleyişi çok net: **Masa → Dosyalık → Klasör → Geçmiş kaydı**

GIT KOMUTLARI

`pwd`

Bu komut bulunduğun klasörün tam yolunu gösterir. Yani “şu anda hangi dizindeyim” sorusunun cevabıdır. Git komutlarını doğru yerde çalıştırıp çalıştırmadığını görmek için kullanılır.

`git init`

Bir klasörü Git deposuna dönüştürür. Bu işlemle birlikte `.git` adında gizli bir klasör oluşur ve bundan sonra yaptığın değişiklikler kayıt altına alınmaya başlar. Yani klasörünü “versiyon kontrolüne” sokmuş olursun.

`ls -la`

Bu komut bulunduğun klasördeki dosyaları listeler. `-l` parametresi detaylı gösterir, `-a` ise gizli dosyaları da ekrana çıkarır. Bu sayede `.git` klasörünü görebilirsin. PowerShell’de bunun karşılığı `ls -Force` şeklindedir.

`git add` ve `git add .`

`git add` belirttiğin dosyaları commit etmeye hazırlık için staging area’ya alır.

- `git add dosya.txt` sadece o dosyayı ekler.
- `git add .` ise bulunduğun klasördeki tüm yeni veya değiştirilmiş dosyaları ekler.

Bu komut, “hangi dosyaları kayda geçirmek istiyorum” sorusunun cevabıdır.

`clear`

Terminal ekranını temizler. Daha düzenli çalışabilmek için kullanılır. Ekranı siler ama aslında geçmişini tamamen yok etmez.

`git status`

Projenin mevcut durumunu gösterir. Hangi dosyaların değiştirilmiş olduğunu, hangilerinin commit edilmeye hazır (staged), hangilerinin yeni (untracked) olduğunu buradan görebilirsin. Commit atmadan önce kontrol amaçlı çok sık kullanılır.

`git commit -m "mesaj"`

Staging area’daki değişiklikleri kalıcı hale getirir ve repository’ye kaydeder. Yanına yazdığın mesaj, bu değişikliğin ne işe yaradığını anlatır. Örneğin `"ilk commit"` ya da `"bug fix"` gibi.

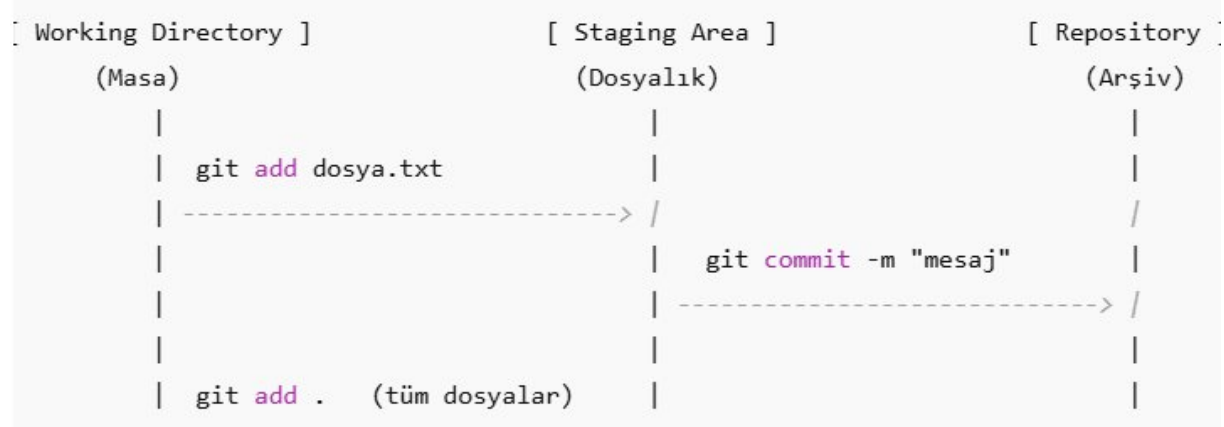
`git log`

Daha önce yapılmış commit’lerin listesini gösterir. Kim, hangi tarihte, hangi mesajla commit yapmış görebilirsin. Kısaca projenin geçmişini takip etmeni sağlar.

Özetle;

1. **pwd** → doğru klasörde miyim kontrol et.
2. **git init** → klasörü Git deposuna dönüştür.
3. **ls -la** → dosyaları gör, `.git` klasörünü kontrol et.
4. **git status** → neler değişmiş bak.

5. `git add .` → tüm değişiklikleri commit'e hazırla.
6. `git commit -m "ilk commit"` → kaydı yap.
7. `git log` → geçmişe bak.
8. `clear` → ekranı temizle.



Açıklama:

- **Working Directory (masa):** Dosyaları burada düzenlersin. Henüz kayda geçmez.
- **Staging Area (dosyalık):** Commit'e hazırladığın dosyaları buraya koyarsın (`git add`).
- **Repository (arşiv):** Commit ile dosyalar kalıcı geçmişe eklenir (`git commit`).

Senaryo Uygulama:

1. `deneme.txt` dosyasını oluşturdun → **Working Directory**
2. `git add deneme.txt` → **Staging Area'ya geçti**
3. `git commit -m "İlk dosya eklendi"` → **Repository'ye kaydedildi**
4. `git log` → Artık geçmişte bu commit görünüyor.

```
git config
```

Git'in ayarlarını yapmak için kullanılan komuttur. Kullanıcı adı, e-posta, varsayılan branch adı, editör, renk ayarları gibi birçok şeyi buradan değiştirebilirsin. Yani Git'in nasıl çalışacağını senin ihtiyaçlarına göre şekillendirmene yarar.

```
git config --system
```

Bu en geniş kapsamlı ayardır. Bilgisayarda Git'i kullanan tüm kullanıcılar için geçerli olur. Yani bilgisayarda birden fazla kullanıcı hesabı varsa, bu ayar onların hepsini etkiler. Bu tür ayarlar genellikle bilgisayarın sistem yöneticisi tarafından yapılır.

Örneğin, bütün kullanıcıların aynı varsayılan metin düzenleyiciyi (nano, vim vs.) kullanmasını istiyorsan, bunu `--system` seviyesinde ayarlarsın. Bu ayarlar genelde bilgisayarın Git'in kurulu olduğu klasördeki yapılandırma dosyasında tutulur.

```
git config --global
```

Bu, sadece senin kullanıcı hesabın için geçerli olan ayardır. Yani bilgisayarda senin kullanıcı adıyla oturum açıldığında bu ayarlar devreye girer. Başka bir kullanıcı aynı bilgisayarı kullanıyorsa, onun için geçerli olmaz.

En çok kullanılan ayarlar burada yapılır. Örneğin, kendi ismini ve e-posta adresini tanımlarken genellikle `--global` kullanırsın. Böylece her yeni proje açtığında Git, otomatik olarak senin adını ve mailini commitlere ekler. Bu ayarlar senin kullanıcı klasörünün içinde `.gitconfig` dosyasında saklanır.

```
git config --local
```

Bu en dar kapsamlı ayardır. Sadece çalıştığın depo (repository) için geçerlidir. Yani sen bir proje klasörünün içindeyken yaptığın `--local` ayarları sadece o projeye uygulanır, başka projeleri etkilemez.

Örneğin, normalde tüm projelerde kendi e-posta adresini kullanıyorsundur ama bir şirkete ait özel bir repo üzerinde çalışırken farklı bir e-posta kullanmak istersin. O zaman `--local` ile bu projeye özel e-posta tanımlayabilirsin. Bu ayarlar da o projenin `.git/config` dosyasında saklanır.

Özet

- **System:** Tüm bilgisayar ve tüm kullanıcılar için geçerli.
- **Global:** Sadece senin kullanıcı hesabın için geçerli.
- **Local:** Sadece bulunduğun proje için geçerli.

Senaryo Uygulama:

- `git config --system core.editor "nano"` → Sistem yöneticisi tüm kullanıcılar için editörü `nano` yaptı.
- `git config --global user.name "Ahmet Yılmaz"` → Ahmet kendi bilgisayar hesabı için adını tanımladı.
- `git config --global user.email "ahmet@gmail.com"` → Ahmet kendi bilgisayar hesabı için kişisel e-postasını tanımladı.
- `git config --local user.email "ahmet@firma.com"` → Ahmet sadece bu repo için farklı bir (şirket) e-posta adresi tanımladı.
- `echo "Merhaba Dünya" > deneme.txt` → **Working Directory** içinde dosya oluşturuldu.
- `git add deneme.txt` → Dosya **Staging Area**'ya geçti.
- `git commit -m "İlk dosya eklendi"` → Dosya **Repository**'ye kaydedildi. (Commit yazarı: Ahmet Yılmaz <ahmet@firma.com>)
- `git log` → Artık geçmişte bu commit görünüyor.

GIT BRANCH

GIT BRANCH KOMUTLARI

```
git log
```

Projedeki commit geçmişini gösterir. Kim, ne zaman, hangi mesajla commit yaptı, hangi değişiklikler oldu görebilirsin. Uzun listeli bir görünüm verir.

```
git log --oneline
```

Commit geçmişini daha kısa ve özet bir şekilde gösterir. Her commit için sadece kısaltılmış hash kodu ve mesaj görünür. Büyük projelerde hızlıca geçmişi taramak için idealdir.

```
git branch -l / git branch --list
```

Yerel dalları listeler. -l ile --list aynı şeydir. Hangi branch'lerin olduğunu ve hangisinde olduğunuzu görebilirsiniz. Yanında * işareti olan, şu an bulunduğunuz dalı gösterir.

```
git branch <branch-ismi>
```

Yeni bir branch oluşturur ama o brancha geçiş yapmaz. Yani sadece dalı açar. Çalışmaya başlamak için ayrıca o dalı seçmen gerekir.

```
git checkout <branch-ismi>
```

Var olan bir branch'e geçiş yapmanı sağlar. Çalışma alanındaki dosyalar seçtiğin branch'in commit'lerine göre güncellenir. Eğer üzerinde kaydedilmemiş değişiklikler varsa Git dal değiştirmeye izin vermez. Günümüzde bunun yerine **git switch** komutu da kullanılabilir.

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

Bu komut commit geçmişini kısa, düzenli ve görsel olarak gösterir.

- **--all**: Tüm branch'lerdeki commit'leri listeler.
- **--decorate**: Commit'lerin yanına branch ve etiket isimlerini yazar.
- **--oneline**: Her commit'i tek satır halinde özetler.
- **--graph**: Commit'leri dalları gösteren çizgilerle grafik gibi çizer.

Bu sayede hangi branch nerede ayrılmış, nerede birleşmiş kolayca görürsün.

```
git merge <branch-ismi>
```

Bu komut, belirttiğin branch'i bulunduğu branch'in içine birleştirir.

- Eğer branch'ler arasında fark yoksa sadece işaretçi ileri alınır (fast-forward).
- Eğer fark varsa Git bir merge commit oluşturur.
- Eğer aynı satırlarda farklı değişiklik varsa conflict (çatışma) çıkar ve senin çözmen gerekir.

Merge, bir branch'teki değişiklikleri ana branch'e veya başka bir branch'e almak için kullanılır.

```
git branch -d <branch-ismi>
```

Bu komut yerel bir branch'i siler.

- **-d**: Güvenli silme yapar, yani o branch merge edilmişse silinir.
- **-D**: Zorla silme yapar, merge edilmemiş olsa bile dal silinir.

Branch üzerinde işin bittiyse ve değişiklikleri ana dala aldıysan -d ile silebilirsin.

Özetle:

- `git log` → commit geçmişini detaylı gösterir.
- `git log --oneline` → commit geçmişini kısa gösterir.
- `git branch --list` → mevcut dalları listeler.
- `git branch <isim>` → yeni dal oluşturur.
- `git checkout <isim>` → o dala geçiş yapar.
- `git log --all --decorate --oneline --graph` → commit geçmişini dallarla birlikte grafik halinde gösterir.
- `git merge <branch>` → branch'i aktif dala birleştirir.
- `git branch -d <branch>` → branch'i güvenli şekilde siler.
- `git log` komutunda aşağıdan yukarıya doğru commitlerin oluşturulma önceliğine göre sıralanırken, `git log -all --decorate -oneline -graph` komutunda ise aşağıdan yukarıya doğru branch'lerin oluşturulma önceliğine göre sıralanır.

Senaryo Uygulama:

- `git log` → Commit geçmişini detaylı şekilde görüntülersin.
- `git log --oneline` → Commit geçmişini kısa ve özet olarak görürsün.
- `git branch --list` → Depodaki mevcut tüm branch'leri listellersin.
- `git branch deneme` → "deneme" adında yeni bir branch oluşturursun.

- `git checkout deneme` → "deneme" branch'ine geçiş yaparsın.
- `git log --all --decorate --oneline --graph` → Tüm commit geçmişini branch'lerle birlikte grafik halinde görürsün.
- `git merge deneme` → "deneme" branch'ini, şu an aktif olduğun branch ile birleştirirsin.
- `git branch -d deneme` → "deneme" branch'ini güvenli bir şekilde silersin.

HEAD KAVRAMI

HEAD'in Temel Tanımı:

Git'te HEAD, senin şu anda bulunduğun yeri gösteren işaretçidir. Git her şeyi commit'ler üzerinden takip eder. Commit'ler projedeki dosyaların fotoğrafları gibidir. HEAD, "şu an hangi commit'in üzerindeyim?" sorusunun cevabıdır.

Çoğu zaman HEAD bir dalı işaret eder. Örneğin, `main` dalındaysan `HEAD` → `main` dalına, `main` de son commit'e işaret eder. Böylece HEAD dolaylı olarak en son commit'i gösterir.

HEAD'in Normal Durumu (Sembolik HEAD):

En çok kullanılan durumda HEAD bir dalı gösterir.

- Yeni commit yaptığında dal ileri kayar, HEAD de onunla birlikte kayar.
- Dal değiştirdiğinde HEAD artık o dalı gösterir.
- Yani HEAD hep bulunduğun dalın en güncel commit'ini işaret eder.

Bu durumda commit atmak güvenlidir çünkü commit'in otomatik olarak o dalın ucuna eklenir.

Detached HEAD (Bağımsız HEAD) Durumu:

Git'te HEAD'in normalde yaptığı gibi bir dalın son commit'ini işaret etmesi yerine, doğrudan bir commit'i (veya tag'i, uzak dalı) işaret ettiği özel durumdur. Yani sen bir dalın ucunda değilsin, geçmişteki ya da belirli bir commit'in tam üzerindesin. Bu durumda dosyaları inceleyebilir, çalıştırabilir ve commit bile yapabilirsin. Ancak yaptığın commit'ler herhangi bir dala bağlı olmadığı için, dal değiştirdiğinde "sahipsiz" kalabilir. Bu yüzden eğer commit'lerin kaybolmasını istemiyorsan, detached HEAD konumundayken mutlaka yeni bir dal açarak o commit'leri bir dala bağlaman gerekir.

HEAD Dosyası:

HEAD bilgisi, projenin içindeki `.git/HEAD` dosyasında saklanır.

- Normalde içinde "ref: refs/heads/main" gibi bir yazı vardır. Bu, HEAD'in `main` dalını işaret ettiğini gösterir.
- Eğer detached moddaysan bu dosyada doğrudan commit'in uzun hash kodu yazar.

Yani HEAD aslında bir metin dosyasında tutulan işaretçidir.

HEAD Ne Zaman Değişir ?

- Yeni commit yaptığında: HEAD'in işaret ettiği dal ileri kayar, HEAD de onunla birlikte hareket eder.
- Dal değiştirdiğinde: HEAD artık yeni dalı işaret eder.
- Eski bir commit'e geçtiğinde: HEAD detached olur ve commit'e bağlanır.
- Reset yaptığında: HEAD farklı commit'e taşınır. Bu işlem bazen sadece HEAD'i etkiler, bazen çalışma dizinini de etkiler.
- Merge veya rebase sırasında: HEAD birleşme sonucu oluşan commit'e ya da yeniden yazılan commit'e taşınır.

HEAD – Staging – Working Directory İlişkisi:

Git'te üç önemli katman vardır:

1. HEAD → En son commit'in kayıtlı hali.
2. Staging Area (Index) → Commit'e eklenmeye hazır dosyalar.
3. Working Directory → Senin bilgisayarındaki dosyaların hali.

HEAD bu üçlüde “tarih”i temsil eder, yani en son kaydedilmiş haldir.

- Reset gibi komutlar bazen sadece HEAD'i değiştirir, bazen staging alanını ve çalışma dizinini de ona göre günceller.
- Bu yüzden bazen dosyaların kaybolmuş ya da eski haline dönmüş gibi görünebilir; aslında HEAD ile senkronize edilmiştir.

HEAD'in Neden Önemli Olduğu:

- Konum göstergesi: Hangi dalda ve hangi commit'te olduğunu gösterir.
- Commit yöneticisi: Yeni commit'ler nereye eklenecek, HEAD bunu belirler.
- Çatışma çözümü: Merge sırasında “bizim taraf” (ours) genellikle HEAD tarafıdır.
- Kurtarma aracı: Yanlış bir işlem yapıldığında HEAD'in hareket geçmişi (reflog) sayesinde eski haline dönebilirsin.

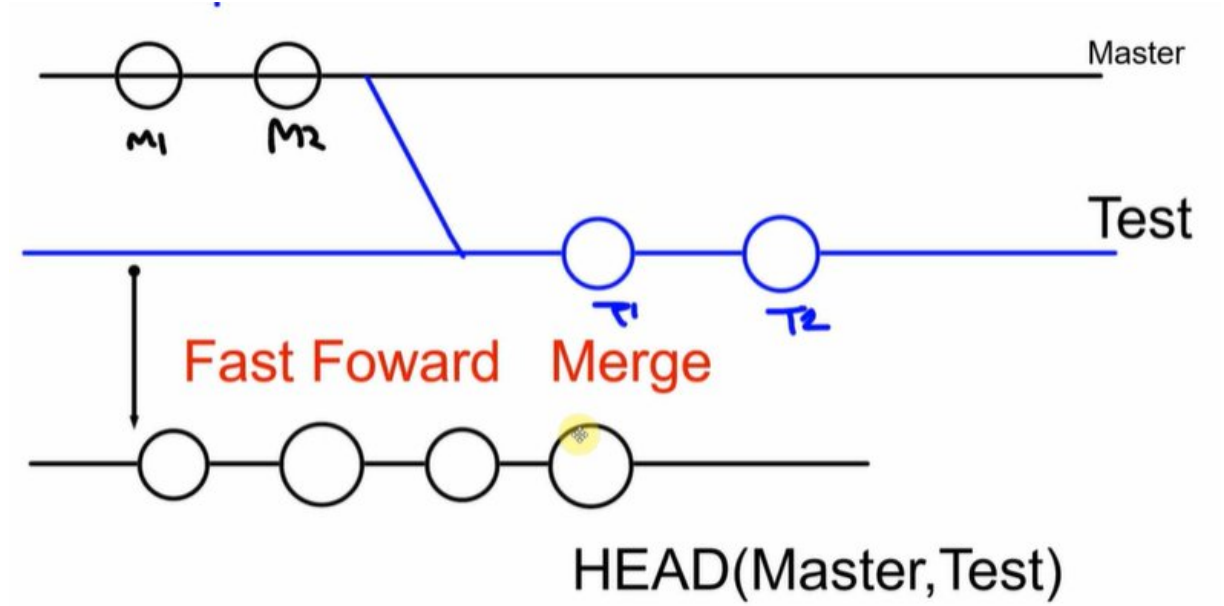
Özet:

HEAD, Git'in “sen şu an buradasın” dediği işaretçidir.

- Normalde bir dalı işaret eder, dal da son commit'i gösterir.
- Bazen detached olur ve doğrudan commit'e bağlanır.
- Commit atma, dal değiştirme, merge, rebase veya reset gibi işlemler HEAD'i sürekli hareket ettirir.
- HEAD'i anlamak, projede nerede durduğunu bilmek ve commit'leri güvenle yönetebilmek için çok önemlidir.

MERGE ÇEŞİTLERİ

Fast Forward Merge:



Fast Forward Merge, iki dal arasında fark olmadığında yapılan en basit birleştirmedir. Örneğin bir feature dalı açtın ve commit yaptın, bu sırada main dalında hiçbir değişiklik olmadı. Böyle bir durumda Git yeni bir merge commit oluşturmaz. Sadece main dalının işaretçisini feature dalının son commit'ine taşır.

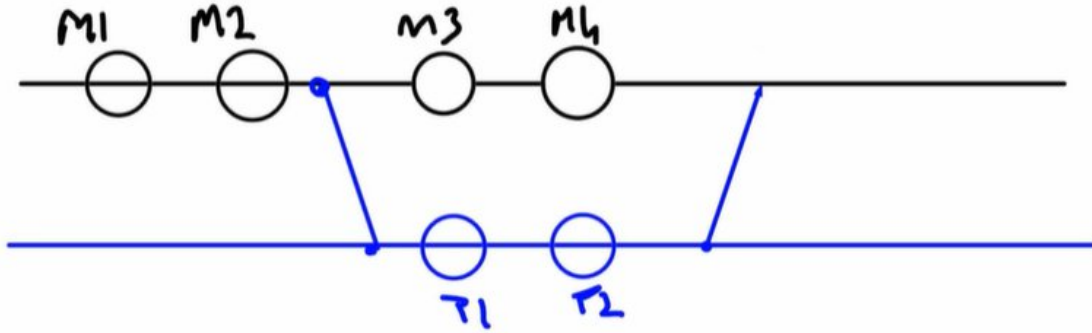
Yani geçmiş çizgisel kalır, sanki hep aynı dalda çalışmışsın gibi görünür.

Avantajı: Geçmiş temiz olur, okunması kolaydır.

Dezavantajı: Ayrı bir dal açıldığını ve o dalda neler yapıldığını geçmişten anlamak zorlaşır.

3-Way Merge:

3 way Merge



3-Way Merge, hem `main` hem de `feature` dalında commit'ler yapıldığında kullanılır. Yani dallar ayrıldıktan sonra iki tarafta da değişiklik olmuştur.

Bu durumda Git üç noktayı karşılaştırır:

- Ortak ata commit (dal ayrılmadan önceki son nokta)
- Aktif dal (örneğin `main`)
- Birleştirilecek dal (`feature`)

Git bu üç kaynağı birleştirir ve yeni bir merge commit oluşturur. Merge commit'in iki ebeveyni vardır: biri `main`, diğeri `feature`.

Avantajı: Hangi dalın hangi değişikliklerle birleştiğini açıkça gösterir, ekip çalışmalarında takip etmek kolay olur.

Dezavantajı: Commit geçmişi dallı budaklı hale gelir, çok sayıda merge commit olursa geçmiş kalabalıklaşabilir.

Karşılaştırma:

- **Fast Forward Merge:** Yeni commit oluşturmaz, sadece işaretçiyi ileri alır. Çizgisel bir geçmiş olur.
- **3-Way Merge:** Yeni bir merge commit üretir. Geçmiş dallanma ve birleşmeleri açıkça gösterir.

Özetle:

- Eğer ana dalda hiç değişiklik yoksa → **Fast Forward Merge** yapılır.
- Eğer her iki dalda da değişiklik varsa → **3-Way Merge** yapılır.

```
git merge --no-ff ( dal-ismi )
```

- Buradaki `--no-ff` parametresi Git'e şunu söyler: "Fast forward mümkün olsa bile, bir merge commit oluştur."

CONFLICT

Conflict Nedir?

Git, birleştirme (merge), yeniden düzenleme (rebase) veya başka dallar üzerinde işlem yaparken dosyaları satır satır karşılaştırır. Eğer aynı dosyanın aynı satırında sen ve arkadaşın farklı değişiklikler yaptıysanız, Git hangi satırın doğru olduğuna kendi başına karar veremez. Bu durumda süreci durdurur ve "conflict" yani çatışma oluşur. Çatışmalar insan müdahalesiyle çözülür: hangi değişikliğin korunacağına veya nasıl birleştirileceğine sen karar verirsin.

Conflict Hangi Durumlarda Olur?

- Sen bir dosyada değişiklik yaptın, arkadaşın da aynı satırda farklı değişiklik yaptıysa.
- Bir taraf dosyayı sildi, diğer taraf değiştirdiyse.
- İki taraf aynı isimle farklı dosya eklediye.
- Bir dal dosyayı yeniden adlandırdı, diğer dal değiştirdiyse.
- İkili dosyalarda (Word, PDF, resim) farklı sürümler oluştuysa.
- Satır sonu veya boşluk farklılıklarından dolayı da bazen çıkar.

Git İçeride Nasıl Çalışır?

Git aslında üç farklı sürümü karşılaştırır:

1. **Base (ortak sürüm):** Dallar ayrılmadan önceki son sürüm.
2. **Ours:** Senin bulunduğun dalın sürümü.
3. **Theirs:** Birleştirmek istediğin dalın sürümü.

Bu üçlü karşılaştırmada, aynı satır farklı şekilde değiştirilmişse Git işin içinden çıkamaz ve çatışma işaretlerini dosyanın içine ekler.

Dosyada Nasıl Görünür?

Git çatışmayı şu işaretlerle gösterir:

```
<<<<<< HEAD
Senin deęiřiklięin
=====
Arkadařının deęiřiklięi
>>>>>> other-branch
```

Burada sen karar verirsin: sadece kendi yazdığını bırakabilirsin, sadece arkadaşınıkini seçebilirsin ya da ikisini birleştirip yeni bir çözüm yazabilirsin. (Bu işaretler arasındaki içerikleri sen inceler ve doğru hali belirlersin. Sonra işaretleri silip dosyanın düzenlenmiş halini kaydedersin.)

Conflict Nasıl Çözülür?

1. Git hangi dosyaların çatıştığını gösterir.
2. Çatışmalı dosyayı açarsın.
3. İşaretlenmiş kısımları inceleyip doğru hali belirlersin.
4. İşaretleri silip dosyayı kaydedersin.
5. Git'e bu dosyanın çözüldüğünü söylersin.
6. İşlemi (merge, rebase, cherry-pick) tamamlayıp süreci bitirirsin.

Conflict'i Önlemek İçin Neler Yapılabilir?

- Aynı dosyanın aynı kısmında çalışmamaya özen göstermek.
- Dalları çok uzun süre ayrı tutmamak, sık sık güncellemek.
- Çalışmayı küçük parçalar halinde yapmak.
- Ortak biçimlendirme kuralları (boşluk, satır sonu, format) kullanmak.
- İkili dosyalarda (Word, resim) bölünmüş çalışma veya Git LFS gibi yöntemler kullanmak.

Çatışma Sonrası Kurtarma Yolları

Eğer yanlış çözüm yaptıysan ya da işin karıştıysa:

- İşlemi tamamen iptal edip başa dönebilirsin.
- Git'in tuttuğu reflog kaydından eski haline geri dönebilirsin.

Özet:

Conflict, Git'in otomatik olarak karar veremediği durumdur. Sen ve arkadaşın aynı dosyada aynı satırı farklı yazdıysanız, Git işaret koyar ve size sorar. Siz birlikte doğru kararı verip dosyayı düzelttikten sonra sürece devam edersiniz. Çatışmaları en aza indirmek için sık sık güncel kalmak ve ortak kurallara uymak önemlidir.

REBASE

Rebase Nedir ?

Git'te **rebase**, bir dalda yaptığın commit'leri alıp başka bir dalın son commit'inin üstüne yeniden yerleştirme işlemidir. Yani senin commit'lerin sanki en baştan beri güncel dalın üzerinde yapılmış gibi görünür.

Bu, aslında commit geçmişini yeniden yazmak demektir. Commit'lerin içeriği aynı kalabilir ama kimlikleri (hash) değişir.

Rebase Ne İşe Yarar ?

- Projedeki commit geçmişini daha temiz ve düz (lineer) hale getirir.
- Uzun süre ayrı kalan bir dalı, ana dalın (main gibi) güncel haliyle hizalamayı sağlar.
- Gereksiz merge commit oluşmasını engeller.
- İstersen commit mesajlarını düzenleyebilir, küçük commit'leri birleştirebilir veya sırasını değiştirebilirsin (interactive rebase).

Rebase Sırasında Ne Olur ?

1. Git, dalının ana daldan ayrıldığı noktayı bulur.
2. Senin bu ayrılıştan sonra yaptığın commit'leri sıraya dizer.
3. Ana dalın en güncel commit'ini taban olarak alır.
4. Senin commit'lerini bu tabanın üstüne tek tek yeniden uygular.
5. Yeni commit'ler oluşur ve her biri yeni hash numarası alır.

Sonuçta içerik aynı kalır ama commit geçmişi **yeniden yazılmış** olur.

Merge İle Farkı ?

- Merge: Dallar birleşir. Geçmişte “bu iki dal burada birleşti” bilgisi korunur. Çoğu zaman ekstra bir merge commit eklenir. Tarih biraz karmaşık ama şeffaf olur.
- Rebase: Senin commit'lerin alınır, diğer dalın sonuna yeniden yazılır. Sanki hiç ayrılmamışsın gibi düz bir çizgi çıkar. Geçmiş daha temiz görünür ama commit hash'leri değişir.

Önemli Not: Dosya içeriği açısından genelde aynı sonuca varırsın, fark commit geçmişindedir.

Ne Zaman Kullanılmalı ?

- Kendi dalında çalışıyorsan ve commit geçmişini daha temiz hale getirmek istiyorsan.
- Pull işlemi yaparken gereksiz merge commit'lerinden kaçınmak istiyorsan (`git pull --rebase`).
- Bir dalı ana dalın en güncel haliyle hizalamak istiyorsan.

Ne Zaman Dikkat Edilmeli ?

- Paylaşılan dallarda rebase yapmak risklidir, çünkü commit hash'leri değişir. Bu durum ekipte karışıklık yaratabilir.
- Rebase genellikle kendi yerel dallarında güvenle kullanılır.

Özet:

Rebase, commit'lerini başka bir dalın en güncel ucunun üstüne yeniden yazarak tarihi **çizgisel ve temiz** hale getiren bir işlemdir. Ancak commit kimliklerini değiştirdiği için dikkatli kullanılmalıdır.

Aşağıdaki görsel merge yapılmış hali: (burada yani aşağıda rebase yapılmadan merge yapılmış ve direkt commit zamanlamalarına göre sıralanmış)

```
MINGW64/c/Users/samet/Desktop/Merge
samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Merge (master)
$ git log
commit 7068dc16accc8df9911afa05803b808989727019 (HEAD -> master)
Merge: 85e3a8c d962d28
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:51:55 2021 +0300

    Merge branch 'test'

commit d962d28a79194a6a2f9e74b0fc22dbd18710333c (test)
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:36:22 2021 +0300

    T2

commit 85e3a8cfa0c18554237332e2889cc2a42b747139
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:21:07 2021 +0300

    M4

commit 2720f18e7a604105e6c37330488fc4d59fa9c315
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:20:45 2021 +0300

    M3

commit d69bf47b1aeb888d3bca9d070713a107e5ce08d7
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:20:12 2021 +0300

    T1

commit 6b153f278071bd141763f1e283a813271c51c567
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:16:10 2021 +0300

    M2

commit d77169af725b81db8aef3d82fc36087818aff407
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:15:41 2021 +0300

    M1

samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Merge (master)
$ git log --all --decorate --oneline
7068dc1 (HEAD -> master) Merge branch 'test'
d962d28 (test) T2
85e3a8c M4
2720f18 M3
d69bf47 T1
6b153f2 M2
d77169a M1

samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Merge (master)
$
```

Aşağıdaki görsel önce rebase yapılmış daha sonra ise merge yapılmış hali: (burada aşağıda ise önce rebase yapıldıktan sonra merge yapılmış ve rebase yapılan branch merge yapılan branch 'in ucuna eklenmiş. Yukarıdaki gibi commit yapılma zamanlamalarına göre değil)

```
MINGW64/c/Users/samet/Desktop/Rebase
samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Rebase (master)
$ git log
commit 063ce657ae466ee0505e34dc01eb769b3a567ef3 (HEAD -> master, test)
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:36:22 2021 +0300

    T2

commit 056060ecbafa4bfbfaed48db64dc3af9d63bbae0
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:20:12 2021 +0300

    T1

commit 85e3a8cfa0c18554237332e2889cc2a42b747139
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:21:07 2021 +0300

    M4

commit 2720f18e7a604105e6c37330488fc4d59fa9c315
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:20:45 2021 +0300

    M3

commit 6b153f278071bd141763f1e283a813271c51c567
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:16:10 2021 +0300

    M2

commit d77169af725b81db8aef3d82fc36087818aff407
Author: Samet Akcalar <sametegitimler@gmail.com>
Date: Tue Jan 26 19:15:41 2021 +0300

    M1

samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Rebase (master)
$ git log --all --decorate --oneline
063ce65 (HEAD -> master, test) T2
056060e T1
85e3a8c M4
2720f18 M3
6b153f2 M2
d77169a M1

samet@DESKTOP-TKG295E MINGW64 ~/Desktop/Rebase (master)
$ |
```

Senaryo Uygulama:

- **git checkout feature** → "feature" branch'ine geçiş yaptın.
- **git rebase main** → Feature branch'teki commit'ler, main'in en güncel commit'inin üzerine yeniden yazılır.
- Git, dalların ayrıldığı noktayı bulur ve feature branch'teki commitleri sıraya dizer.
- Main dalının en son commit'ini taban olarak alır.
- Senin commitlerini bu tabanın üstüne tek tek yeniden uygular. (Her commit yeni hash alır.)
- Eğer çatışma çıkarsa, çözümledikten sonra `git rebase --continue` ile devam edersin.

- Eğer işler karışır, `git rebase --abort` ile işlemi iptal edip eski haline dönersin.
- Eğer tek bir commit sorunluysa, `git rebase --skip` ile o commit'i atlayabilirsin.
- Rebase bittiğinde commit geçmişi düz (lineer) ve temiz görünür.

ALIAS

Alias Nedir?

Git'te alias, uzun veya sık kullanılan komutların kısa ve kolay hatırlanabilir bir adla kullanılmasını sağlayan bir özelliktir. Normalde `git status` yazman gerekirken, alias sayesinde sadece `git st` yazabilirsin. Alias, Git'in kendi yapılandırma dosyalarına kaydedilir ve Git bir komut çalıştırıldığında önce bunun bir alias olup olmadığını kontrol eder. Eğer tanımlı bir alias bulursa, onu asıl komutla değiştirip öyle çalıştırır.

Alias'ın Amacı:

- Her gün kullanılan uzun komutları kısaltarak zaman kazandırmak.
- Tekrar tekrar hatırlanması zor olan uzun seçenekleri tek bir ad altında toplamak.
- Takım içinde ortak kısayollar oluşturarak komutların tutarlı kullanılmasını sağlamak.
- Hata yapma ihtimalini azaltmak, çünkü her zaman aynı hazır alias kullanılabilir.

Alias nerelerde tanımlanır ?

Git ayarları üç katmanda tutulur. Alias da bu katmanlarda tanımlanabilir:

1. **System düzeyi:** Bilgisayardaki tüm kullanıcılar için geçerli olur. Sistem yöneticisi seviyesinde ayarlanır.
2. **Global düzey:** Sadece senin kullanıcı hesabın için geçerli olur. Kişisel alias'lar genellikle buraya eklenir.
3. **Local düzey:** Sadece belirli bir repository için geçerlidir. Yalnızca o projeyi etkiler.

Öncelik sırası **Local > Global > System** şeklindedir. Yani aynı isimde bir alias üç yerde tanımlıysa, en yakındaki (local) alias çalışır.

Özet:

Git'te alias, sık kullanılan veya uzun komutları kısaltmaya yarayan bir mekanizmadır. Local, global veya system seviyesinde tanımlanabilir. Alias sayesinde komutlar daha kısa, daha hızlı ve daha güvenli hale gelir. Basit alias'lar tek komut için kullanılırken, shell alias'ları daha karmaşık işlemleri gerçekleştirebilir. Alias'lar özellikle günlük akışta büyük kolaylık sağlar ama dikkatli tasarlanmalıdır.

GIT Alias Kullanım Örnekleri ve Nasıl Kullanıldığı:

1) Global Alias: Global alias sadece senin kullanıcı hesabında geçerli olur.

Örneğin sık kullanılan `status` komutunu kısaltalım:

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

Artık `git status` yerine sadece şu yazılır:

```
git st
```

Global alias'lar senin `~/.gitconfig` dosyana kaydedilir.

2) Local Alias: Eğer bir alias'ı sadece belli bir projede kullanmak istiyorsan local olarak tanımlarsın.

```
git config --local alias.co checkout
```

Artık o repository içinde `git co` yazarsan `git checkout` çalışır. Ama bu alias başka repo'da geçerli olmaz. Local alias'lar ilgili repo'nun `.git/config` dosyasına kaydedilir.

3) System Alias (tüm bilgisayar için): System alias bütün kullanıcılar için geçerli olur. Daha çok sistem yöneticileri tanımlar.

```
git config --system alias.br branch
```

Artık tüm kullanıcılar `git br` yazınca `git branch` çalışır. System alias'lar genellikle `/etc/gitconfig` gibi sistem dosyalarına kaydedilir. Bu komutu çalıştırmak için yönetici (root / admin) yetkisi gerekebilir.

- Eğer alias sadece kısa bir ad veriyorsa ve içinde boşluk yoksa tırnağa ihtiyaç yoktur eğer alias içine birden fazla komut veya boşluk içeren seçenekler yazıyorsan, bunları tek bir bütün olarak saklamak için tırnak kullanmak zorundasın.

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

```
git config --global alias.lg "log --oneline --graph --decorate --all"
```

```
MINGW64:/c:/Users/lenovo/OneDrive/Masaüstü/EXAMPLE
lenovo@DESKTOP-72HJDNK MINGW64 ~/OneDrive/Masaüstü/EXAMPLE (yeniBranch)
$ git config alias.adog "log --all --decorate --oneline --graph"

lenovo@DESKTOP-72HJDNK MINGW64 ~/OneDrive/Masaüstü/EXAMPLE (yeniBranch)
$ git adog
* d248257 (HEAD -> yeniBranch) Version 1.6
* 35dd096 Version 1.5
| * 7530e3e (master) Version 1.4
| * 0277aac Version 1.3
|/
* 89dda0c Version 1.2
* 5ea30be Version 1.1

lenovo@DESKTOP-72HJDNK MINGW64 ~/OneDrive/Masaüstü/EXAMPLE (yeniBranch)
$ git config alias.adg "log --all --decorate --graph"

lenovo@DESKTOP-72HJDNK MINGW64 ~/OneDrive/Masaüstü/EXAMPLE (yeniBranch)
$ git adg
* commit d248257976c88c22f4042e4889af69801c215128 (HEAD -> yeniBranch)
| Author: Kasım Teke <kasm-teke45@hotmail.com>
| Date: Fri Sep 26 12:29:55 2025 +0300
|
| Version 1.6
* commit 35dd096c8a34b3fd5fed722d6a622ab9e851dd04
| Author: Kasım Teke <kasm-teke45@hotmail.com>
| Date: Fri Sep 26 12:29:39 2025 +0300
|
| Version 1.5
|
| * commit 7530e3e7c68141eca3b5ebcc174b8297209c5155 (master)
| | Author: Kasım Teke <kasm-teke45@hotmail.com>
| | Date: Fri Sep 26 12:29:05 2025 +0300
| |
| | Version 1.4
| |
| | * commit 0277aacc3e9ee13ecf32190844fb444fad35c403
| | | Author: Kasım Teke <kasm-teke45@hotmail.com>
| | | Date: Fri Sep 26 12:28:48 2025 +0300
| | |
| | | Version 1.3
| | |
|/
lenovo@DESKTOP-72HJDNK MINGW64 ~/OneDrive/Masaüstü/EXAMPLE (yeniBranch)
$ |
```

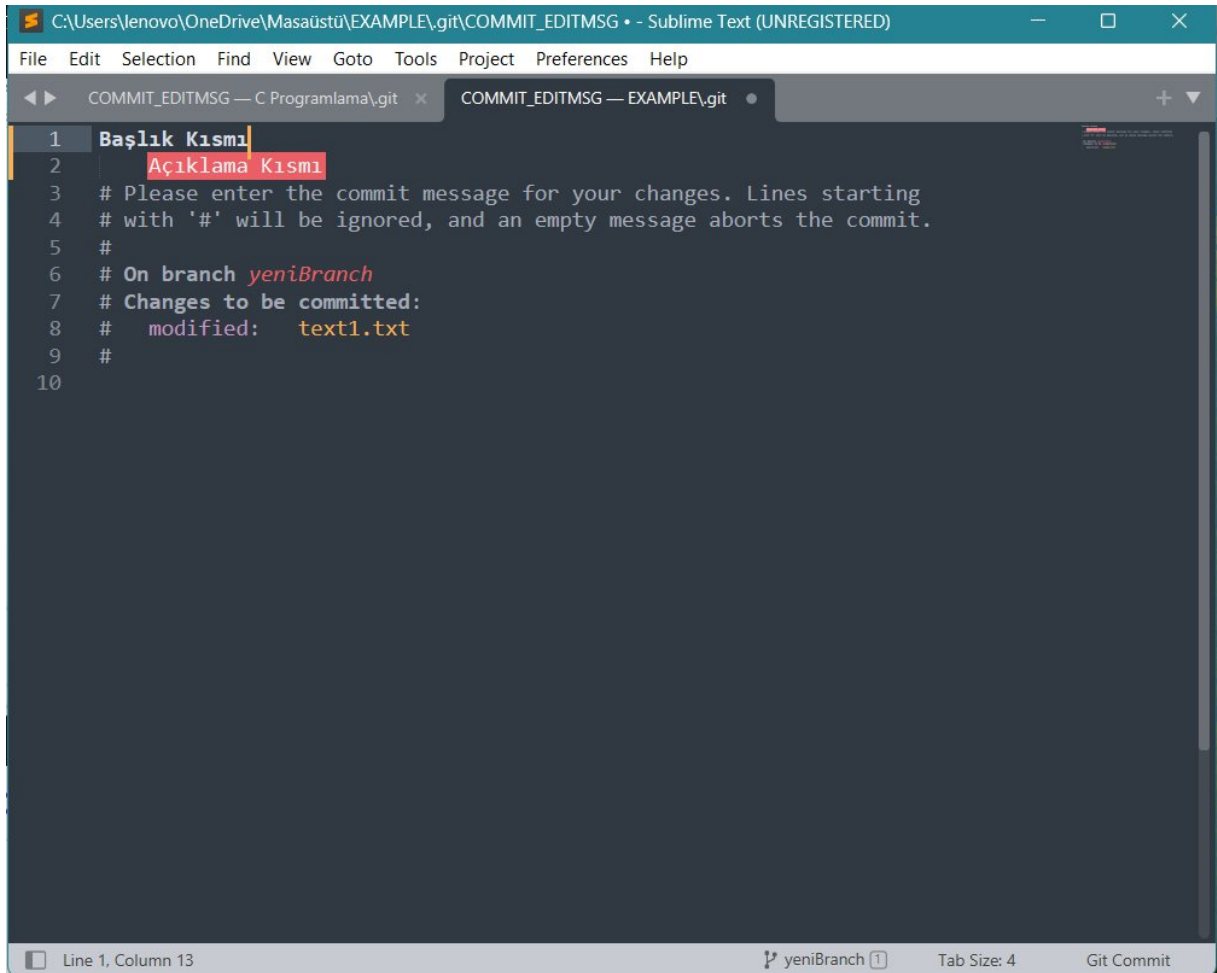
Senaryo Uygulama:

- **git config --global alias.st status** → Global seviyede git st kısayolu tanımlandı. Artık git st yazınca git status çalışır.
- **git config --local alias.co checkout** → Local seviyede git co kısayolu tanımlandı. Bu sadece o repo için geçerlidir.
- **git config --system alias.br branch** → System seviyesinde git br kısayolu tanımlandı. Tüm kullanıcılar için geçerlidir.
- **git config --global alias.lg "log --oneline --graph --decorate --all"** → Global seviyede git lg kısayolu tanımlandı. Artık commit geçmişini tek satır, grafik ve dallarla birlikte görürsün.
- Alias'ların önceliği Local > Global > System şeklindedir. Aynı isim üç yerde de tanımlıysa, en yakındaki alias çalışır.
- Basit alias'larda tırnak gerekmez (git config --global alias.st status), ama boşluk içeren alias'larda tırnak şarttır (git config --global alias.lg "log --oneline --graph --decorate --all").

İyi Bir Commit Nasıl Atılmalı ?

- Her bir commit bir amaca hizmet etmelidir.
- Commitleri siz değil başkası okuyacakmış gibi yazınız.
- Konu satırlarını kısa ve öz tutun. Fazla uzun commit mesajları yazmayınız.
- Tamamlanmamış değişiklikleri commit atmamaya özen gösteriniz.
- Commit mesajlarınızı İngilizce yazmaya özen gösteriniz.
- Neyi, neden ve nasıl olduğunu açıklamak için açıklama alanını kullanın.
- Konuyu ve açıklamayı boş bir satırla ayırın.
- Commitlerin sonuna nokta koymayınız.

Açıklama ve başlık kısmı aşağıdaki gibidir. Bu kısım ise commit açılırken sadece git commit yazarak açılıyor.



```
1 Başlık Kısmı
2 Açıklama Kısmı
3 # Please enter the commit message for your changes. Lines starting
4 # with '#' will be ignored, and an empty message aborts the commit.
5 #
6 # On branch yeniBranch
7 # Changes to be committed:
8 #   modified:   text1.txt
9 #
10
```

GIT IGNORE

.gitignore Nedir ?

.gitignore, bir yazılım projesinde kullanılan özel bir dosyadır. Bu dosya, Git'e hangi dosya ve klasörleri hiç dikkate almaması gerektiğini söyler. Git normalde bir proje klasörünün içindeki her şeyi takip etmek ister, çünkü versiyon kontrolü mantığı budur: zamanla dosyalarda yapılan değişiklikleri kaydetmek. Ancak bazı dosyalar vardır ki, bunların takip edilmesi mantıklı değildir. Bunlar ya tekrar

üretilebilen, ya kişiye özel, ya da güvenlik riski taşıyan dosyalardır. `.gitignore` sayesinde bu dosyalar Git'in radarına hiç girmez.

Basitçe düşünersek: `.gitignore`, “Bu dosyaları projede sayma, yokmuş gibi davran” talimatıdır.

.gitignore Neden Gereklidir ?

Bir proje sadece kaynak kodlardan ve gerçekten gerekli yapı dosyalarından oluşmalıdır. Ama geliştirme sürecinde birçok dosya yan ürün olarak ortaya çıkar.

- Geçici dosyalar: Örneğin programın çalışması sırasında oluşturulan log dosyaları veya hata kayıtları.
- Derleme çıktıları: Kaynak koddan üretilen `bin/`, `obj/`, `dist/`, `out/` klasörleri. Bunlar zaten tekrar derlenerek elde edilebilir.
- Kişisel ayar dosyaları: Bir IDE'nin (örneğin Visual Studio Code veya IntelliJ) oluşturduğu `.vscode/`, `.idea/` klasörleri. Bunlar kişisel tercihler içerir, takım için anlamlı değildir.
- Sistem dosyaları: İşletim sisteminin kendiliğinden oluşturduğu `.DS_Store` (Mac) veya `Thumbs.db` (Windows) gibi dosyalar.
- Gizli bilgiler: Şifreler, API anahtarları, bağlantı bilgileri içeren `.env` dosyaları.

Bu dosyalar commit edilirse:

1. Depo şişer ve boyutu gereksiz yere büyür.
2. Takımda gereksiz çatışmalar çıkar. Örneğin her geliştiricinin IDE ayarları farklı olabilir.
3. Gizli bilgilerin paylaşılması ciddi güvenlik açığı yaratır.

`.gitignore`, işte bu sorunları çözmek için kullanılır.

.gitignore Nasıl Çalışır ?

Git'in dosyaları takip etme mantığı üç aşamalıdır: çalışma alanı (working directory), geçiş alanı (staging area) ve commit.

- Bir dosya ilk oluşturulduğunda, Git bunu “takip edilmeyen” (untracked) olarak görür.
- Eğer bu dosyayı `git add` komutuyla eklemeye çalışırsan, staging area'ya alınır ve commit edilebilir.
- Ama `.gitignore` dosyasında bu dosyanın adı veya deseni yazılıysa, Git bu dosyayı hiç görmezden gelir. Yani onu staging area'ya bile almaz.

Burada kritik bir nokta var: Eğer bir dosya daha önce commit edildiyse, `.gitignore` onu yok saymaz. Yani `.gitignore`, sadece henüz commit edilmemiş dosyalara etki eder. Daha önce commit edilmiş dosyanın takibini bırakmak istiyorsan, ayrıca `git rm --cached` ile o dosyayı indeks dışına çıkarman gerekir.

.gitignore Nereye Konur ?

- Genelde proje kök klasöründe bulunur ve tüm proje için geçerli olur.
- Alt klasörlerde de `.gitignore` dosyası oluşturulabilir. Bu durumda sadece o klasör ve altındaki dosyalar için geçerlidir.
- Takımın birlikte çalıştığı kurallar `.gitignore` dosyasında tutulur ve commit edilir. Böylece herkes aynı ignore kurallarını kullanır.

Kişisel ihtiyaçlar için ise:

- Global bir `.gitignore` dosyası oluşturabilirsin. Örneğin kendi bilgisayarındaki `.DS_Store` veya `Thumbs.db` dosyalarının her projede yok sayılmasını istiyorsan, `~/.gitignore_global` oluşturulur.
- Sonrasında `git config --global core.excludesfile ~/.gitignore_global` komutuyla bu dosya global ignore listesi haline getirilir.

.gitignore Kuralları Nasıl Yazılır ?

Kurallar satır satır yazılır. Her satır bir kuraldır.

- `dosya.txt` → sadece bu dosya yok sayılır.
- `*.log` → tüm `.log` uzantılı dosyalar yok sayılır.
- `klasör/` → bu klasör ve içindekiler yok sayılır.
- `**/cache/` → hangi klasörde olursa olsun tüm `cache` klasörleri yok sayılır.
- `!dosya.txt` → daha önce ignore edilmiş olsa bile bu dosya hariç tutulur, yani takip edilmeye devam eder.

Kural yazarken `*`, `?`, `[]`, `**` gibi joker karakterler kullanılabilir.

- `*` herhangi bir karakter dizisini temsil eder.
- `?` tek bir karakteri temsil eder.
- `**` birden fazla klasör seviyesini temsil edebilir.

Gerçek Örnekler:

- Node.js projelerinde: `node_modules/` klasörü her zaman ignore edilir, çünkü bu klasör paket yöneticisiyle zaten yeniden kurulabilir.
- Python projelerinde: `__pycache__/` ve `.venv/` klasörleri ignore edilir.
- .NET projelerinde: `bin/` ve `obj/` klasörleri yok sayılır.
- Çapraz platformda: `.DS_Store` ve `Thumbs.db` gibi sistem dosyaları ignore edilir.

.gitignore Sınırlamaları:

- `.gitignore`, yalnızca takip edilmeyen dosyalara etki eder. Daha önce commit edilmiş dosyalar için işe yaramaz.
- Eğer yanlışlıkla gizli bir dosya commit ettiysen, `.gitignore` o dosyayı geçmiş commitlerden silemez. Onu kaldırmak için ya geçmiş commitleri düzenlemen ya da özel araçlarla geçmişini temizlemen gerekir.
- Git boş klasörleri takip etmez. Eğer bir klasörün projede bulunmasını istiyorsan ama içeriğinin boş kalması gerekiyorsa, genellikle `.gitkeep` adında boş bir dosya koyulur.

Sonuç:

`.gitignore`, bir projeyi temiz, güvenli ve düzenli tutmak için hayati bir dosyadır. Git'e hangi dosya ve klasörlerin yok sayılması gerektiğini söyler. Böylece log dosyaları, derleme çıktıları, kişisel ayarlar ve gizli bilgiler repoya girmemiş olur. Kurallar basit desenlerle yazılır, gerekirse istisna tanımlamak için `!` işareti kullanılır. Daha önce commit edilmiş dosyalar için ayrıca takipten çıkarma işlemi yapılması gerekir.

`.gitignore` ve Önceden Commit Edilmiş Dosyalar:

`.gitignore` sadece takip edilmeyen dosyalar üzerinde etkilidir. Yani:

- Bir dosya daha önce commit edilmişse Git onu zaten izliyordur.
- Sen o dosyanın adını `.gitignore` dosyasına yazsan bile, Git onu yok saymaz. Çünkü Git'in mantığı şu: "Bu dosya projenin bir parçası olarak geçmişte kayıt altına alınmış. O yüzden ben onu izlemeye devam etmeliyim."

Kısacası `.gitignore`, geçmiş commit'leri değiştirmez.

Ne Gibi Sorunlar Olur ?

Diyelim `.env` dosyanı yanlışlıkla commit ettin. Daha sonra `.gitignore` dosyasına `.env` yazdın.

- Git bu dosyayı takip etmeye devam eder.
- Sen `.env` içinde yeni bir şey yazdığında Git bunu değişiklik olarak algılar ve `git status` çıktısında gösterir.
- Yani `.gitignore` işe yaramıyor gibi görünür.

Bu Durumda Ne Yapılmalı ?

Böyle bir dosyayı gerçekten ignore etmek istiyorsan, Git'e "bu dosyayı artık takip etme" demelisin. Bunun için dosyayı indeksten çıkarmak gerekir.

Bunun yolu:

1. Dosyayı indeksten çıkar ama bilgisayardan silme:

```
git rm --cached dosya.txt
```

Bu komut, dosyanın Git tarafından izlenmesini bırakır ama dosya senin bilgisayarında kalmaya devam eder.

2. Dosyayı `.gitignore` dosyasına yaz:
Artık Git bu dosyayı tekrar izlemeye çalışmaz.
3. Commit at:
Bu değişiklikle birlikte, proje geçmişinde o dosya kalır ama bundan sonra izlenmez.

Daha Önceki Commitlerin Varlığı

- Eğer dosyanın daha önceki commitlerde bulunması sorun değilse, sadece yukarıdaki işlem yeterlidir. Bundan sonra proje temiz şekilde devam eder.
- Ama eğer dosya gizli bilgi içeriyorsa (örneğin şifre veya API anahtarı) ve geçmişte görünmesi sorun yaratıyorsa, o zaman daha ileri bir işlem gerekir: geçmiş commitlerden dosyayı

tamamen silmek. Bunun için git filter-branch veya git filter-repo gibi araçlarla geçmiş temizlenir. Sonrasında da şifre/anahtar mutlaka değiştirilir (çünkü geçmişte sızmış olur).

Özet:

- `.gitignore`, sadece commit edilmemiş dosyalara etki eder.
- Eğer bir dosya commit edilmişse `.gitignore` dosyasına yazsan bile Git onu takip etmeyi sürdürür.
- Çözüm: `git rm --cached` ile dosyayı Git'in takibinden çıkarmak ve `.gitignore` dosyasına eklemek.
- Gizli dosyalar yanlışlıkla commit edildiyse, gerekirse geçmişten de temizlemek gerekir.

GIT DIF

Git Diff Nedir ?

`git diff`, Git tarafından takip edilen dosyalardaki farklılıkları gösteren bir komuttur. Bu komut, iki farklı durum arasında satır satır karşılaştırma yapar ve hangi satırların silindiğini, hangi satırların eklendiğini net bir şekilde ortaya koyar. Çıktıda:

- - işareti olan satırlar silinmiş satırları,
- + işareti olan satırlar eklenmiş satırları gösterir.

Bu sayede, proje üzerinde yapılan değişiklikleri çok net bir şekilde görebilirsin.

Git'in Üç Katmanı ve Git Diff İlişkisi

Git'te dosyaların bulunduğu üç temel alan vardır:

1. Çalışma dizini (Working Directory): Bilgisayarındaki gerçek dosyaların güncel hali. Editörde açıp üzerinde çalıştığın dosyalar buradadır.
2. Staging alanı (Index): Bir sonraki commit için hazırlanmış dosyaların listelendiği alan. `git add` komutu ile dosyaları buraya gönderirsin.
3. HEAD: Şu anda bulunduğun dalın en son commit'ini temsil eder.

`git diff`, bu üç alan arasında farklı bakış açılarıyla kıyaslama yapmana olanak tanır.

Git Diff Hangi Durumlarda Neyi Gösterir ?

- Hiçbir parametre olmadan (`git diff`): Bu komutu tek başına çalıştırırsan Git sana çalışma dizinindeki (working directory) değişikliklerle staging alanındaki (index) dosyaları karşılaştırır.
- `git diff --staged` veya `git diff --cached`: Bu durumda Git sana staging alanındaki dosyalarla HEAD (yani en son commit edilmiş hali) arasındaki farkı gösterir.
- `git diff HEAD`: Bu komut çalışma dizinindeki ve staging alanındaki tüm değişiklikleri HEAD ile kıyaslar. Yani proje en son commit edilmiş haliyle şu an senin bilgisayarındaki hali arasındaki farkları toplu olarak görürsün.

- `git diff commit1 commit2`: Bu komut iki commit arasındaki farkları gösterir. Yani geçmişte belirli iki nokta arasındaki değişiklikleri görebilirsin.
- `git diff branch1 branch2`: Bu komut iki dalın (branch) en son commit'lerini karşılaştırır. Yani dallar arasındaki güncel farkları görürsün.
- `git diff main...feature`: Bu en çok kafa karıştıran kısımdır. Üç nokta (...) olduğunda Git, önce main ve feature dallarının ortak atasını bulur. Sonra feature dalının bu noktadan bugüne kadar yaptığı değişiklikleri gösterir.

Çıktıyı Okuma Mantığı

`git diff` çıktısı dosya dosya gelir. Her dosya için:

- Başlık kısmında hangi dosyanın karşılaştırıldığı yazar.
- @@ -10,5 +10,6 @@ gibi satırlar, değişikliğin dosyanın neresinde olduğunu gösterir.
- - ile başlayan satırlar silinmiş satırlardır.
- + ile başlayan satırlar eklenmiş satırlardır.
- Başında boşluk olan satırlar değişmemiş satırlardır, sadece bağlam sağlamak için gösterilir.

Eğer dosya yeni eklenmişse “new file mode” yazar, dosya tamamen silinmişse “deleted file mode” yazar. Binary dosyalarda satır satır karşılaştırma yapılamadığı için sadece “binary files differ” gibi bir bilgi çıkar.

Neden Önemlidir ?

- Commit atmadan önce kontrol: Hangi satırların commit'e gireceğini görüp hata yapmaktan kaçınabilirsin.
- Kod inceleme: Başkasının yaptığı değişiklikleri satır satır inceleyebilirsin.
- Hata ayıklama: Bir bug çıktıysa, hangi satırların değiştiğini hızlıca bulabilirsin.
- Refaktör ve büyük değişiklikler: Dosya adlarının değişmesi, kod bloklarının taşınması gibi durumlarda diff çıktısı sana net bilgi verir.

Özet

`git diff`, iki farklı nokta arasındaki farkları satır satır gösteren çok güçlü bir komuttur. Çalışma dizinindeki, staging alanındaki ve commit edilmiş dosyalar arasındaki farkları farklı kombinasyonlarla görebilirsin. Çıktıda + eklenen, - silinen satırı ifade eder. Commit atmadan önce son kontrol için, kod incelemelerinde ve hata ayıklamada çok sık kullanılır. Ayrıca boşluk ve satır sonu farklarını filtreleyerek daha temiz sonuç almak mümkündür.

Senaryo Uygulama:

1) `git diff`

👉 Senaryo:

- `main.c` dosyasında yeni bir fonksiyon yazdın.
- Henüz `git add main.c` yapmadın.

🔪 Çalıştırdığında:

```
bash
```

 Kodu kopyala

```
git diff
```

Çıktıda sadece **staging'e alınmamış değişiklikler** görünür. Yani senin `main.c` dosyasında yaptığın son değişiklikler listelenir.

2) `git diff --staged` (veya `git diff --cached`)

👉 Senaryo:

- `utils.c` dosyasına bir fonksiyon ekledin.
- `git add utils.c` yaptın ama commit etmedin.

🔪 Çalıştırdığında:

```
bash
```

 Kodu kopyala

```
git diff --staged
```

```
# veya
```

```
git diff --cached
```

Çıktıda commit'e girmek üzere **staged edilmiş değişiklikler** görünür. Yani `utils.c` 'ye eklediğin fonksiyon satır satır listelenir.

Not: `--staged` ile `--cached` tamamen aynı işi yapar.

3) `git diff branchIsmi1 branchIsmi2`

👉 Senaryo:

- `main` dalında commit C5 var.
- `feature` dalında commit F3 var.

📌 Çalıştırdığında:

```
bash
```

[Kodu kopyala](#)

```
git diff main feature
```

Çıktıda C5 ile F3 commit'leri kıyaslanır. Yani iki dalın en son halleri arasındaki farkları görürsün.

Kullanım amacı: Bir dalı başka bir dal ile birleştirmeden önce hangi farklar olduğunu görmek.

4) `git diff commit1 commit2`

👉 Senaryo:

- Commit geçmişinde: C1 → C2 → C3 → C4 var.
- Sen C2 ile C4 arasındaki farkı görmek istiyorsun.

📌 Çalıştırdığında:

```
bash
```

[Kodu kopyala](#)

```
git diff C2 C4
```

Çıktıda sadece C2'den C4'e kadar olan değişiklikler listelenir.

Kullanım amacı: Belirli bir zaman aralığında projeye nelerin eklendiğini incelemek.

RESTORE

Git Restore Nedir ?

`git restore`, Git'te bir dosyayı geri almak için kullanılan bir komuttur. Daha önce bu işi `git checkout` ve `git reset` ile yapıyorduk ama bu komutlar karıştı. Git restore, sadece dosyaları geri almak için özel olarak çıkarıldı.

Git Restore Ne İşe Yarar ?

Bir dosya Git'te üç farklı yerde bulunur:

1. Çalışma dizini: Senin bilgisayarındaki dosyanın anlık hali.
2. Staging alanı: Commit'e girmek üzere hazırlanmış hali.
3. HEAD: En son commit edilmiş hali.

Bazen yanlışlıkla dosyada değişiklik yaparsın, bazen yanlışlıkla staging'e eklersin. Git restore bu durumlarda dosyayı geri almanı sağlar.

Hangi Durumlarda Kullanılır ?

- Çalışma dizinindeki değişiklikleri silmek:
Dosyayı editörde değiştirdin ama beğenmedin. Git restore ile onu staging alanındaki ya da commit edilmiş haline geri döndürebilirsin.
- Staging alanına alınmış dosyayı geri çıkarmak:
Yanlışlıkla `git add` yaptın. Restore ile dosyayı staging'den çıkarabilirsin ama dosya senin bilgisayarında kalır.
- Eski commit'teki halini getirmek:
Bir dosyanın önceki commit'teki halini geri getirmek isteyebilirsin. Git restore o commit'teki içeriği alıp senin dosyana yazar.
- Parça parça geri almak:
Dosyanın sadece belirli satırlarını geri almak istersen restore bunu parça parça yapmana izin verir.
- Çatışma çözmek:
Merge sırasında aynı satır farklı kişiler tarafından değiştirilmişse, restore ile “benim tarafım” veya “karşı taraf” değişikliğini seçebilirsin.

Özet:

`git restore`, dosyaları geri almak için kullanılan anlaşılır bir komuttur.

- Editördeki değişiklikleri silebilirsin.
- Staging'den dosya çıkarabilirsin.
- Eski commit'teki dosyayı geri getirebilirsin.
- Çatışmalarda hangi tarafı seçeceğine karar verebilirsin.

Yani restore sayesinde yanlış yaptığında projeni güvenle eski haline döndürebilirsin.

Senaryo Uygulama:

Bir proje üzerinde çalışıyorsun ve `notlar.txt` dosyasında değişiklik yaptın.

• Dosyada birkaç satırı değiştirdin ama sonra bu değişikliklerden memnun kalmadın. Dosyayı en son commit edilmiş haline geri döndürmek için `git restore notlar.txt` komutunu kullandın. Sonuç: Çalışma dizinindeki değişiklikler silindi, dosya commit'teki haline döndü.

Bir başka durumda yanlışlıkla `git add notlar.txt` yaptın ve dosyayı staging alanına gönderdin. Commit atmadan önce aslında staging alanında olmaması gerektiğini fark ettin. Bunun için `git restore --staged notlar.txt` komutunu kullandın. Sonuç: Dosya staging alanından çıkarıldı ama yaptığın değişiklikler çalışma dizininde durmaya devam etti.

Daha sonra projede yeni bir dal üzerinde çalışırken `ayarlar.json` dosyasında değişiklik yaptın. Bu değişikliklerin sadece bir kısmını geri almak istedin. `git restore -p ayarlar.json` komutuyla parça parça geri almayı seçtin. Böylece sadece istemediğin satırlar geri alındı, diğer değişiklikler kaldı.

Bir gün merge sırasında çatışma çıktı. Aynı satır sen ve takım arkadaşın tarafından değiştirilmişti. Git sana “hangi tarafı istiyorsun?” diye sordu. Sen kendi tarafındaki değişiklikleri atmak için `git restore --source=HEAD -- bir_dosya.txt` komutunu kullandın. Böylece dosya senin en son commit'teki haline geri döndü.

Özet:

- `git restore dosya`: Çalışma dizinindeki değişiklikleri siler, dosyayı commit haline döndürür.
- `git restore --staged dosya`: Dosyayı staging alanından çıkarır, ama değişiklikler çalışma dizininde kalır.
- `git restore -p dosya`: Dosyanın sadece belirli kısımlarını geri almanı sağlar.
- `git restore --source=HEAD dosya`: Çatışma durumunda dosyayı HEAD'deki (son commit) haline döndürür.

GIT CHECKOUT

Git Checkout Nedir?

`git checkout`, Git'te hem dal (branch) yönetimi hem de dosya yönetimi için kullanılan temel komutlardan biridir. Bu komut sayesinde bir projede farklı dallar arasında geçiş yapılabilir veya belirli dosyalar farklı commit'lerden ya da dallardan geri getirilebilir. Başka bir deyişle, `git checkout` hem “ben şu dala geçmek istiyorum” demektir, hem de “şu dosyayı şu hâline döndür” anlamında kullanılabilir.

Git Checkout Neden Gereklidir?

Bir yazılım projesinde iki temel ihtiyacı karşılar:

1. Dal Değiştirme:
 - Farklı dallarda çalışmak için bir dalı terk edip diğerine geçmek gerekir.
 - Örneğin “main” dalında çalışırken, yeni bir özellik geliştirmek için feature/login dalına geçebilirsiniz.

2. Dosya Yönetimi:

- Yanlışlıkla değiştirdiğin veya silmek istemediğin bir dosyayı geri almak için kullanılır.
- Ayrıca başka bir commit veya daldaki bir dosyanın hâlini almak için de kullanılabilir.

Bu iki temel işlev, yazılım geliştirme sürecinde çok sık karşılaşılan ihtiyaçlardır.

Git Checkout Nasıl Çalışır?

Git'in mantığı üç katmandan oluşur: çalışma alanı (working directory), geçiş alanı (staging area) ve commit edilmiş geçmiş (HEAD). git checkout komutu bu katmanlar üzerinde değişiklik yapar:

- Eğer dal değiştirme için kullanılırsa:
 - HEAD göstergesini yeni dala taşır.
 - Staging area ve çalışma dizini de yeni dalın commit'ine uyarlanır.
- Eğer dosya geri alma için kullanılırsa:
 - Dosyayı belirtilen commit'teki ya da index'teki hâline döndürür.
 - Böylece yanlışlıkla yapılan değişiklikler geri alınır.

Burada önemli nokta şudur: Eğer bir dosya daha önce commit edilmişse, git checkout onu silemez. Sadece yeni değişiklikleri görmezden gelir veya geri alır.

Git Checkout ile Dal İşlemleri:

- `git checkout main` → main dalına geçer.
- `git checkout -b yeni-dal` → yeni bir dal oluşturur ve ona geçer.
- `git checkout -` → bir önce bulunduğun dala geri döner.
- `git checkout -f main` → değişiklikler olsa bile zorla main dalına geçer (dikkatli kullanılmalı).

Uzak dalı almak için:

- `git checkout -b dev origin/dev` → uzak repodaki origin/dev dalını takip eden yerel bir dev dalı oluşturur.

Git Checkout ile Dosya İşlemleri:

- `git checkout -- dosya.txt` → dosyayı staging area'daki hâline döndürür.
- `git checkout HEAD -- dosya.txt` → dosyayı en son commit'teki hâline döndürür.
- `git checkout main -- dosya.txt` → başka bir daldaki dosyayı kendi dalına alır.
- `git checkout commitID -- dosya.txt` → belirli bir commit'teki hâliyle dosyayı getirir.

Parça parça geri almak için:

- `git checkout -p -- dosya.txt` → dosyanın değişikliklerini satır satır sorar, seçilen parçaları geri alır.

Çakışma (merge conflict) durumlarında:

- `git checkout --ours dosya.txt` → kendi dalındaki hâlini alır.
- `git checkout --theirs dosya.txt` → diğer dalın hâlini alır.

Detached HEAD Durumu:

Eğer `git checkout <commitID>` yazarsan, HEAD doğrudan bir commit'i gösterir. Bu durumda bir dala bağlı olmazsın ve yaptığın commit'ler sahipsiz kalır. Buna **detached HEAD** denir.

Eğer bu hâlde çalışıp commit yaptıysan, onları kaybetmemek için yeni bir dal açman gerekir:

- `git checkout -b yeni-dal`

Böylece yaptığın commit'ler korunur.

Git Checkout ve Yeni Komutlar:

Git'in yeni sürümlerinde `checkout` komutunun işleri iki ayrı komuta bölünmüştür:

- **git switch:** Dal değiştirmek için.
- **git restore:** Dosya geri almak için.

Yine de `git checkout` hâlâ geçerlidir ve her iki işi de yapar.

Git Checkout Sınırlamaları:

- Eğer bir dosya daha önce commit edildiyse, `git checkout` onu geçmişten silemez. Sadece değişiklikleri geri alabilir.
- Yanlış dalda çalışmaya başlarsan ve commit yaparsan, dal bağlı değilse (detached HEAD) commit'ler boşa gidebilir. Bu durumda yeni dal açarak onları korumak gerekir.
- Zorla geçiş (`-f`) yerel değişikliklerini kaybettirebilir, dikkatle kullanılmalıdır.

Sonuç:

`git checkout`, Git'in en güçlü komutlarından biridir. Hem dallar arasında geçiş yapmaya hem de dosyaları geri almaya yarar.

- Dal değiştirirken HEAD'i yeni dala taşır ve çalışma alanını ona göre günceller.
- Dosya geri alırken ise belirtilen commit'teki ya da index'teki hâliyle dosyayı çalışma alanına kopyalar.
Modern Git'te işleri daha net ayırmak için `git switch` ve `git restore` önerilse de, `git checkout` hâlâ günlük işlerde yaygın ve önemlidir.

GIT RESET

Git Reset Nedir ?

`git reset`, Git'te bir dalın işaret ettiği commit'i geri veya başka bir commit'e taşımak için kullanılan güçlü bir komuttur. Yani aslında dalın ucunu oynatır. Bunun yanında staging area (index) ve çalışma klasörü (working directory) üzerinde de etki eder. Hangi alanları değiştireceği ise seçilen moda bağlıdır. Böylece commit'leri geri alabilir, dosyaları “unstage” yapabilir veya tamamen temizleyebilirsin.

Git Reset Neden Gereklidir ?

Bir yazılım projesinde farklı durumlarda `git reset` kullanma ihtiyacı doğar:

- Yanlış commit'i geri almak: Hatalı bir commit yapıldığında ama değişikliklerin kaybolmaması istendiğinde.
- Dosyaları unstaged yapmak: Yanlışlıkla staging area'ya alınan dosyaları geri almak için.
- Geçmişten yeniden düzenlemek: Birkaç commit'i birleştirmek veya silmek için.
- Dal ile uzak repository'yi eşitlemek: Origin/main gibi uzak bir dalın son durumuna birebir dönmek için.

Bu ihtiyaçlar özellikle geliştirme sürecinde çok sık karşılaşılr.

Git Reset Nasıl Çalışır ?

Git üç ana katman üzerinde çalışır:

- **HEAD (commit geçmişi):** Üzerinde bulunduğun dalın en son commit'i.
- **Index (staging area):** Bir sonraki commit'e girecek dosyaların hazırlandığı alan.
- **Working directory (çalışma klasörü):** Dosyaların gerçek hali, diskte gördüğün kopya.

`git reset`, önce HEAD'i seçilen commit'e taşır. Ardından, kullanılan moda göre index ve çalışma klasörünü bu commit'e göre eşitler ya da olduğu gibi bırakır.

Git Reset Modları:

--soft

- Sadece HEAD değişir.
- Index ve çalışma klasörü dokunulmaz.
- Son commit geri alınır ama değişiklikler staged halde kalır.
- Kullanım: Commit mesajını düzeltmek veya birkaç commit'i yeniden birleştirmek istediğinde.

--mixed (varsayılan)

- HEAD değişir, index yeni HEAD'e eşitlenir.
- Çalışma klasörü olduğu gibi kalır.
- Son commit geri alınır, değişiklikler unstaged hale gelir.
- Kullanım: Yanlış commit'i geri almak ama dosyaların çalışma alanında kalmasını istemek.

--hard

- HEAD, index ve çalışma klasörü yeni commit'e eşitlenir.
- Son commit ve yerel değişiklikler tamamen silinir.
- Kullanım: Her şeyi temizleyip belirli bir commit'e birebir dönmek istediğinde.
- Dikkat: Geri dönüş zor olabilir, reflog ile kurtarma gerekebilir.

--merge

- HEAD'i taşır, index'i yeni HEAD'e eşitler.
- Çalışma klasöründeki değişiklikleri korumaya çalışır, çakışma olursa işlemi durdurur.
- Kullanım: Başarısız merge işlemlerinden kurtulurken.

--keep

- HEAD'i taşıır.
- Çalışma klasöründeki değişikliklere dokunmaz. Eğer ezilme riski varsa reset işlemini yapmaz.
- Kullanım: Dal ucunu geri almak ama çalışma klasörünü kesinlikle bozmamak istediğinde.

Git Reset ile Dosya Bazlı İşlemler:

Reset sadece HEAD'i geri almak için kullanılmaz, dosya bazlı da çalışabilir.

- `git reset HEAD -- dosya.txt` → Dosyayı staged durumdan unstaged durumuna alır. İçeriğe dokunmaz.
- `git reset <commit> -- dosya.txt` → Dosyanın index'teki halini, seçilen commit'teki sürüme döndürür.
- `git reset -p` → Parça parça (hunk bazında) unstaging yapılmasını sağlar.

Git Reset ile Kullanılan Notasyonlar:

- `HEAD^` → Bir önceki commit.
- `HEAD~3` → 3 commit geri.
- `branch~2` → Belirli bir dalda 2 commit geri.
- `HEAD@{1}` → HEAD'in reflog'daki bir önceki durumu.
- `ORIG_HEAD` → Büyük işlemlerden önceki konum.

Bu notasyonlar sayesinde reset işlemi esnek ve güçlü hale gelir.

Git Reset ve Diğer Komutlarla Karşılaştırma:

- **Reset:** Dal ucunu değiştirir, index ve çalışma klasörünü moduna göre etkiler. Geçmişini yeniden yazar.
- **Checkout / Switch:** Sadece başka dala veya commit'e geçiş yapar, geçmişini yazmaz.
- **Restore:** Dosyaları index veya commit'ten geri getirir, dal ucunu değiştirmez.
- **Revert:** Hatalı commit'in etkilerini tersine alan yeni bir commit üretir, geçmişini bozmadan düzeltme yapar.

Kural: Yayınlanmış (push edilmiş) commit'lerde reset tehlikelidir, bunun yerine revert tercih edilmelidir.

Reset'in Sınırlamaları:

- Paylaşılan commit'lerde reset yapmak takımın geçmişini bozar.
- `--hard` kullanıldığında yerel değişiklikler tamamen silinir.
- Reset, untracked dosyalara dokunmaz. Onları temizlemek için `git clean` gerekir.
- Yanlış reset sonrası tek kurtarıcı `git reflog` olabilir.

Sonuç:

`git reset`, Git'in en güçlü ama en dikkatli kullanılması gereken komutlarından biridir.

- Küçük düzeltmeler için `--soft` veya `--mixed`.
- Tam temizlik için `--hard`.
- Dosya bazlı "unstage" için `paths` kullanımı.
Doğru şekilde kullanıldığında geçmiş düzenler, çalışma ortamını toparlar ve hataları hızlıca düzeltir. Ancak paylaşılan geçmişte risklidir, bu yüzden `revert` daha güvenli bir alternatiftir.

Git Reset Komutları:

```
git restore --staged dosya.txt
```

- Dosyayı staging area'dan çıkarır.
- Yani `git add` yapmıştın ama `commit` atmadan önce vazgeçtin → bu komut dosyayı staging'den siler.
- Working directory'deki değişiklikler kalır, sadece staged olmaktan çıkar.

```
git reset --hard dosya.txt
```

- Burada kritik nokta: `git reset --hard` tek dosya adı almaz `commit index` 'i alır, sadece HEAD'i komple geri sarar.
- Yani bu komutun doğru hali:
- `git reset --hard`
- Etki: Staging area ve working directory'deki tüm değişiklikler silinir, HEAD'teki son commit'e geri dönersin.
- Bu yüzden dosya adı yazarsan hata verir veya beklediğin gibi çalışmaz.

```
git reset --mixed dosya.txt
```

- Benzer şekilde, `git reset --mixed` de tek dosya adı almaz `commit index` 'i alır, **HEAD pointer + index** üzerinde çalışır.
 - `--mixed` default moddur. Yani:
 - `git reset --mixed <commit-id>`
→ HEAD'i belirtilen commit'e alır, **staging area sıfırlanır**, ama working directory dokunulmaz.
 - Tek dosya için staging'den çıkarmak istersen, zaten `git restore --staged dosya.txt` kullanılır.
-

```
git reset --soft commit-id
```

- Yine tek dosya adı almaz, commit id alır.
- Kullanımı:
- `git reset --soft <commit-id>`
- Etki: HEAD geri sarılır ama **staging area korunur**. Yani commit'i geri alırsın ama değişiklikler staged olarak durmaya devam eder.

Fark özet:

- `--soft` → Commit geri alınır, dosyalar staged kalır.
 - `--mixed` → Commit geri alınır, staging sıfırlanır, dosyalar working directory'de kalır.
 - `--hard` → Commit geri alınır, staging sıfırlanır, working directory de silinir → tamamen temizlenir.
-

```
rm dosya.txt
```

- Bu Git komutu değil, **Linux/PowerShell** dosya silme komutudur.
 - Etki: Dosyayı **diskten siler** (working directory'den kaybolur).
 - Ama Git açısından dosya hâlâ staged olabilir → yani sonraki `git status`'te "deleted" olarak görünür.
 - Eğer bunu commit edersen, Git repo'sundan da silinmiş olur.
-

Genel Özet

- `git restore --staged dosya.txt` → Dosyayı staging'den çıkar (ama bilgisayarda durur).
- `git reset --soft` → Commit geri al, staging'i koru. (dosya bazlı değil, commit bazlıdır)
- `git reset --mixed` → Commit geri al, staging'i temizle, dosya değişiklikleri working directory'de kalsın.
- `git reset --hard` → Commit geri al, staging'i temizle, working directory'yi de temizle (dosyaları son commit'e döndür).
- `rm dosya.txt` → Çalışma klasöründen dosyayı fiziksel olarak sil.

Senaryo Uygulama:

Bir proje üzerinde çalışıyorsun ve `deneme.txt` dosyasında değişiklik yaptın.

- Önce bu dosyayı staging alanına göndermek için `git add deneme.txt` yaptın.
- Sonra `git commit -m "deneme dosyası güncellendi"` ile commit ettin.

Bir süre sonra bu commit'in yanlışlıkla atıldığını fark ettin ve farklı senaryolara göre geri almak istedin:

1. `git reset --soft HEAD ~ 1`
Bu komutu kullandığında son commit geri alındı ama dosya staging alanında kalmaya devam etti. Yani commit iptal edildi fakat dosya sanki yeniden commit edilmeye hazır bekliyor. Bu sayede sadece commit mesajını değiştirmek veya küçük ekleme yapmak istiyorsan, tekrar commit atabilirsin.
2. `git reset --mixed HEAD ~ 1`
Bu durumda commit geri alındı ve dosya staging alanından da çıkarıldı. Artık dosya sadece çalışma dizininde değişmiş halde duruyor. Tekrar commit etmek istersen önce `git add` ile yeniden staging alanına göndermen gerekir. Bu, en sık kullanılan reset türüdür.
3. `git reset --hard HEAD ~ 1`
Bu komutu çalıştırdığında commit geri alındı, staging alanındaki değişiklikler silindi ve çalışma dizinindeki dosya da tamamen son commit'in haline döndü. Yani tüm yaptığın değişiklikler kayboldu. Bu nedenle çok dikkatli kullanılması gerekir.

Bir başka durumda, `deneme.txt` dosyasında değişiklik yaptın ama henüz commit etmedin. Dosyanın en son commit edilmiş haline geri dönmek istedin. Bunun için `git checkout -- deneme.txt` kullandın. Sonuç: Çalışma dizinindeki değişiklikler silindi ve dosya commit'teki haline döndü.

Daha sonra projede farklı bir dala geçmen gerekti. `feature-x` adında bir dal vardı. `git checkout feature-x` yazarak bu dala geçtin. Çalışma dizinindeki dosyalar bu dalın içeriğiyle değişti ve artık o dal üzerinde geliştirme yapmaya başladın.

REMOTE REPOSITORY

Remote repository, yani uzak depo, projenin internette veya ağ üzerinde saklanan kopyasıdır. Senin bilgisayarında bir local repository vardır, bu uzak depo onun uzaktaki versiyonudur. Bir ekiple çalışırken herkes kendi bilgisayarında çalışır ama değişikliklerin ortak bir yerde toplanması gerekir. İşte o ortak nokta uzak depodur.

Uzak depo sayesinde:

- Sen commit'lerini kendi bilgisayarında yaparsın, sonra `push` ile uzak depoya gönderirsin.
- Arkadaşın kendi bilgisayarında commit yapar, o da `push` ile gönderir.
- Sen `pull` yaptığında onun yaptığı değişiklikleri alırsın.
- Yani herkesin katkıları bir yerde birleşmiş olur.

Uzak depo aynı zamanda bir yedek gibidir. Bilgisayarında proje silinse bile uzak depoda kopyası durur. Ayrıca bir projede birden fazla uzak depo da olabilir, mesela `origin` ve `upstream` gibi.

GITHUB

GitHub, uzak depoları barındıran bir platformdur. Yani GitHub senin projeni internette tutar. Git olmadan GitHub olmaz çünkü GitHub, Git depoları için bir hizmettir.

GitHub sadece dosyaları saklamakla kalmaz, aynı zamanda ekip çalışmasını kolaylaştırır. Örneğin:

- Takım üyeleri aynı proje üzerinde çalışabilir.
- Pull request ile biri değişiklik önerdiğinde diğerleri inceleyebilir.
- Issue sistemi ile hatalar ve görevler takip edilebilir.
- Proje yönetimi için araçlar sunar.
- Açık kaynak projeler burada paylaşılır, başka projelere katkı yapılabilir.
- GitHub profili, bir yazılımcı için portföy görevi görür.

Remote Repository ve GitHub Arasındaki İlişki:

Remote repository aslında kavramsal bir şeydir. Yani “projenin uzaktaki kopyası” demektir. GitHub ise bu uzak depoyu barındıran bir yerdir. Başka servisler de vardır (GitLab, Bitbucket gibi) ama en popüler olanı GitHub'dır.

Özet:

- Remote repository, projenin uzaktaki kopyasıdır.
- GitHub, bu kopyayı internette saklamayı ve ekipçe çalışmayı sağlayan bir platformdur.
- Git bir araçtır, GitHub ise bu aracı kullanmana yardımcı olan bir hizmettir.

SENARYO: Ortak Proje Geliştirme

Sen ve arkadaşın bir yazılım projesi yapıyorsunuz. Herkesin kendi bilgisayarında Git kurulu ve local repository'si var. Ama birlikte çalışabilmek için bir ortak noktaya ihtiyacınız var. Bunun için GitHub'da bir repository açıyorsunuz.

1. İlk Başlangıç

Sen bilgisayarında `git init` komutu ile proje klasörünü local repository'ye dönüştürüyorsun. Dosyalar ekliyorsun, commit atıyorsun. Arkadaşın da kendi bilgisayarında aynı şeyi yapıyor. Bu noktada siz ikiniz de projeye sahipsiniz ama birbirinizin yaptığı şeylerden haberiniz yok.

Sen diyelim ki projenin A bölümünde çalışmaya başlıyorsun. Kod yazıyor, commit atıyorsun. Arkadaşın ise B bölümünde çalışıyor. O da kendi bilgisayarında commit'ler atıyor. Yani siz ikiniz de aynı projenin farklı kısımlarında ama bağımsız bir şekilde ilerliyorsunuz.

2. GitHub'da Ortak Depo Oluşturma

Sen GitHub'a girip bir repository oluşturuyorsun. Bu, sizin remote repository yani uzak deponuz oluyor. Sen bilgisayarındaki local repository'yi bu uzak depoya bağlıyorsun. Daha sonra commit'lerini `git push` ile GitHub'a gönderiyorsun.

Artık GitHub'da projenin bir kopyası var. Bu kopya sizin ortak buluşma noktanız haline geliyor.

3. Arkadaşının Katılması

Arkadaşın GitHub'daki bu projeyi kendi bilgisayarına indiriyor (`git clone`). Böylece onun bilgisayarında da aynı proje hazır hale geliyor. O da kendi local repository'sine sahip oluyor.

Artık sen ve arkadaşın, aynı uzak depoya bağlı iki ayrı local repository ile çalışıyorsunuz.

4. Paralel Çalışma

Sen A bölümünde dosyaları geliştiriyorsun, commit atıyorsun. Arkadaşın B bölümünde başka dosyalar üzerinde çalışıyor, o da commit atıyor. Siz farklı yerlerde değişiklik yaptığınız için işler sorunsuz gidiyor.

Senin commit'lerin önce sadece senin bilgisayarında duruyor. Arkadaşının commit'leri de sadece onun bilgisayarında duruyor. Henüz ikiniz de birbirinizin yaptığı işleri görmüyorsunuz.

5. Değişikliklerin Paylaşılması

Arkadaşın yaptığı commit'leri GitHub'a gönderiyor (`git push`). Sen de biraz sonra projeyi güncel tutmak için `git pull` yapıyorsun.

- `git pull` ile GitHub'daki yeni commit'ler senin bilgisayarına indiriliyor.
- Git bu commit'leri senin kendi commit'lerinle birleştiriyor.
- Senin A bölümünde yaptıkların bozulmadan duruyor.
- Arkadaşının B bölümünde yaptıkları da senin projenin içine ekleniyor.

Sonuç: Senin bilgisayarında hem senin yaptığın A bölümü değişiklikleri hem de arkadaşının yaptığı B bölümü değişiklikleri birleşmiş oluyor.

Bu noktadan sonra sen A bölümünde çalışmaya aynen devam edebilirsin.

6. Çatışma Durumu (Conflict)

Diyelim ki bir gün sen de arkadaşın da aynı dosyanın aynı satırında değişiklik yaptınız. Arkadaşın önce push yaptı, commit'leri GitHub'a gitti. Sen push yapmak istediğinde Git sana izin vermiyor ve "remote repository senden ileride, önce pull yapmalısın" diyor.

Sen `git pull` yaptığında Git karşılaştırma yapıyor ve şunu görüyor:

- Aynı satırda hem senin değişikliğin var hem de arkadaşının.

Bu durumda Git kararı sana bırakıyor. Dosyanın içine işaretler koyuyor. Sen dosyayı açıyorsun ve iki farklı versiyonu görüyorsun. Şimdi karar zamanı:

- Sadece senin yazdığın kalsın,
- Sadece arkadaşının yazdığı kalsın,
- İkisini birleştirip yeni bir çözüm yaz.

Dosyayı düzenledikten sonra kaydediyorsun ve commit ediyorsun. Artık proje yeniden ortak hale gelmiş oluyor.

7. Döngünün Devamı

Bu süreç proje bitene kadar defalarca tekrar ediyor:

- Sen commit yapıyorsun, push ediyorsun.
- Arkadaşın commit yapıyor, push ediyor.
- Siz birbirinizin commit'lerini `pull` ile çekiyorsunuz.
- Çoğu zaman Git değişiklikleri otomatik birleştiriyor.
- Bazen çatışma oluyor, siz manuel olarak çözüyorsunuz.

Her yeni commit en sonunda GitHub'daki remote repository'de toplanıyor.

8. Sonuç

- Senin bilgisayarında projenin bir kopyası var.
- Arkadaşının bilgisayarında bir kopyası var.
- GitHub'da da en güncel ve tam hali var.

Yani proje üç yerde güvenle duruyor. Bilgisayarınız bozulsa bile GitHub'da yedeği bulunuyor.

Özet:

Remote repository sizin ortak çalışma alanınız. GitHub bu uzak depoyu barındıran bir platform. Siz kendi bilgisayarlarınızda bağımsız olarak çalışıyorsunuz ama yaptıklarınızı push ve pull ile ortak noktada birleştiriyorsunuz. Çakışma yoksa işler otomatik birleşiyor. Çakışma varsa Git size gösteriyor ve siz karar veriyorsunuz.

(**Bir Başka Örnek:** Bir projede çalışırken, güncel kodları GitHub'dan çekmek için bilgisayarındaki projeyi silmene gerek yoktur. GitHub'dan çektiğin değişiklikler, var olan dosyalarının üstüne kaba bir şekilde yazılmaz. Bunun yerine Git, senin bilgisayarındaki proje ile GitHub'daki projeyi karşılaştırır. Bu karşılaştırmayı dosya dosya, hatta satır satır yapar.

Eğer senin üzerinde çalıştığın dosyalarda değişiklik yoksa, GitHub'daki güncel dosyaları getirip ekler. Eğer senin de değişiklik yaptığın dosyalar varsa, o zaman Git ince birleştirme işlemi yapar. Yani senin yazdıklarını korur, GitHub'dan gelen yenilikleri de dosyanın içine yerleştirir.

Burada iki ihtimal vardır:

- Senin değiştirdiğin satırlar ile GitHub'dan gelen satırlar farklı yerlere denk gelmişse, Git bunları otomatik birleştirir. Sen hem kendi değişikliklerini hem de başkalarının yaptığı yenilikleri birlikte görürsün.
- Eğer aynı satır hem senin tarafında hem de GitHub tarafında farklı şekilde değiştirilmişse, Git karar veremez. Bu durumda “conflict” çıkar. Git dosyanın içine işaretler koyar ve sana her iki sürümü de gösterir. Sen bu noktada seçimini yaparsın: ya kendi değişikliğini bırakır, ya arkadaşınınkini kabul eder, ya da ikisini birleştirip yeni bir çözüm yazarısın.

Bu yüzden projeyi güncellemek için bilgisayarındaki klasörü silmene gerek yok. Git zaten senin dosyalarını koruyarak çalışır. Çekme işlemi (pull) sadece GitHub'daki yeni commit'leri bilgisayarına getirir ve senin commit'lerinle birleştirir. Senin yaptığın işler yok olmaz, sadece gerekirse birleştirme sırasında senin onayına ihtiyaç duyar.

Kısaca: GitHub'dan güncel projeyi çekmek, senin bilgisayarındaki projeyi silmez. Sadece farklılıkları indirir, seninkilerle birleştirir. Çatışma yoksa otomatik birleşir, çatışma varsa karar sana bırakılır.)

GITHUB TEMEL KOMUTLAR

```
git push origin master
```

Bu komut, yereldeki **master** dalındaki commit'leri uzak depoya (**origin**) gönderir. Böylece senin bilgisayarında yaptığın değişiklikler GitHub'daki master dalına eklenmiş olur.

```
git pull origin master
```

Bu komut, uzak depodaki master dalındaki en güncel değişiklikleri indirir ve kendi lokal dalına birleştirir. Böylece takım arkadaşlarının yaptığı güncellemeleri kendi bilgisayarına alırsın.

```
git remote add origin (repo-adresi)
```

Bu komut, lokal projen ile uzak depo arasında bağlantı kurar. **origin** kısa adını vererek her seferinde uzun URL yazmadan push/pull yapabilirsin.

```
git remote rename origin upstream
```

Bu komut, daha önce eklenmiş uzak deponun kısa adını değiştirir. Örneğin **origin** adını **upstream** yapabilirsin. Bu özellikle fork senaryolarında kullanılır.

```
git clone <repo-linki>
```

Bu komut, uzak bir Git deposunun tam kopyasını bilgisayarına indirir. Projenin tüm dosyaları, commit geçmişi ve dalları yerel ortama alınır. Ayrıca uzak depo ile bağlantı `origin` adıyla kaydedilir.

Senaryo Uygulama:

- **git clone <https://github.com/kullanici/proje.git>** → GitHub'daki uzak depoyu kendi bilgisayarına indirdin. Projenin dosyaları, commit geçmişi ve dalları yerelde hazır hale geldi. Ayrıca uzak depo bağlantısı `origin` adıyla tanımlandı.
- **git pull origin master** → Klonladıktan sonra uzak depodaki `master` dalında yeni commit'ler varsa onları kendi lokal kopyana aldın. Böylece senin bilgisayarındaki proje, GitHub'dakiyle tamamen eşitlendi.
- Projeye yeni bir dosya ekledin → Working Directory
- **git add .** → Yeni dosyayı staging alanına aldın.
- **git commit -m "Yeni dosya eklendi"** → Bu dosyayı lokal repository'ye kaydettin.
- **git push origin master** → Yaptığın yeni commit'i uzak depoya gönderdin. Artık değişiklik GitHub'da da görünüyor.
- Daha sonra takım arkadaşın yeni bir özellik ekledi → Uzak Repository güncellendi
- **git pull origin master** → Arkadaşının yaptığı güncellemeleri kendi lokal kopyana indirdin ve kodun güncel kalmasını sağladın.
- Sen bir dosyada değişiklik yaptın → Commit ettin → **git push origin master** → Yaptığın son değişiklikler yine GitHub'a gönderildi, herkes tarafından erişilebilir oldu.
- Daha sonra projedeki uzak depo adını değiştirmek istedin → **git remote rename origin upstream** → Artık uzak bağlantıyı `origin` yerine `upstream` ismiyle kullanabilirsin.

FORK & PULL REQUEST

Fork:

- Amaç: Başkasına ait bir projeyi kendi hesabına kopyalamak.
- Neden? Orijinal projede yazma yetkin olmasa bile, kendi alanında özgürce değişiklik yapabilmek.
- Sonuç: Artık senin hesabında, istediğin gibi oynayabileceğin bağımsız bir kopya var.

Pull Request (PR)

- Amaç: Fork'un ya da kendi dalının üzerinde yaptığın değişiklikleri, orijinal projeye veya ana dala önermek.
- Neden? Senin yaptığın geliştirmelerin ekibe katılabilmesi, başkaları tarafından incelenmesi ve kabul edilmesi.

- Sonuç: Bir “birleştirme talebi” oluşturmuş oluyorsun. Proje yöneticileri kodunu gözden geçirip uygun görürlerse projeye dahil ediyorlar.

Özetle:

- Fork = “Projeyi kendi alanıma kopyalayayım, özgürce çalışayım.”
- Pull request = “Yaptığım bu değişiklikleri projeye eklemeyi öneriyorum.”

Fork – Clone – Pull Arasındaki Fark

- Fork: Uzak depoyu kendi hesabında kopyalarsın → web üzerinden yapılan bir işlemdir.
- Clone: Bir uzak depoyu (ister orijinal, ister fork’un) bilgisayarına indirirsin → yerelde çalışmak için.
- Pull: Daha önce klonladığın projenin güncel değişikliklerini bilgisayarına çekersin → projeyi senkron tutmak için.

Basit benzetme:

- Fork = Kitapçının rafındaki kitabı kendi kitaplığında saklamak için aynı kitaptan bir kopya almak.
- Clone = Kitaplığındaki kitabı kendi masana getirmek.
- Pull = Kitapçının yeni baskısını alıp masandaki kopyana eklemek.

SEMANTİK VERSİYONLAMA KAVRAMI

Semantik Versiyonlama Nedir ?

Semantik versiyonlama, yazılım sürümlerine verilen numaraların anlamlı bir kurala göre belirlenmesidir. Amaç, sürüm numarasını gören kişinin yazılımda ne tür değişiklikler olduğunu anlayabilmesidir.

Sürüm Numarasının Yapısı

Bir sürüm numarası şu şekilde yazılır:

MAJOR.MINOR.PATCH

Örneğin 2.4.7

- MAJOR (ana sürüm): Büyük ve uyumsuz değişiklikler için.
- MINOR (alt sürüm): Yeni özellikler eklenince ama uyumluluk bozulmadığında.
- PATCH (yama sürüm): Küçük hata düzeltmeleri ve iyileştirmeler için.

MAJOR (Ana Sürüm)

Yazılımda köklü değişiklikler yapıldığında artırılır.
Önceki sürümü kullananların kodları artık çalışmayabilir.
Örneğin 1.0.0 → 2.0.0

MINOR (Alt Sürüm)

Yeni özellikler eklenir ama eski özellikler çalışmaya devam eder.
Uyumluluk bozulmadığı için kullanıcılar sorunsuz geçiş yapabilir.
Örneğin 2.3.0 → 2.4.0

PATCH (Yama Sürüm)

Küçük hatalar düzeltilir veya ufak iyileştirmeler yapılır.
Yeni özellik gelmez, sadece var olan sorunlar giderilir.
Örneğin 2.4.7 → 2.4.8

Neden Önemli ?

- Kullanıcı sürüm numarasına bakarak değişiklik türünü anlayabilir.
- Ekip içinde standart bir iletişim dili sağlar.
- Paket yöneticileri uyumlu sürümleri doğru şekilde yükler.
- Yazılımın hangi sürümünde ne kadar risk olduğunu anlamayı kolaylaştırır.

Günlük Hayattan Örnek

Bir telefon uygulaması düşün:

- Uygulama tamamen yenilendi, menüler değişti → **MAJOR**
- Uygulamaya karanlık mod eklendi → **MINOR**
- Çökme hatası düzeltildi → **PATCH**

Kısacası:

- **MAJOR:** Büyük, uyumsuz değişiklik
- **MINOR:** Yeni özellik ama uyumlu
- **PATCH:** Hata düzeltme

Senaryo Uygulama:

- Yazılım projesini yayınladın. İlk sürümün numarası 1.0.0 olarak belirlendi. Bu, kullanıcılara “artık stabil bir sürüm var” mesajı verdi.
- Zamanla kullanıcılar küçük bir hata buldu. Sen bu hatayı düzelttin, yazılımda başka bir şey değişmedi. Bu yüzden sürüm numarası 1.0.0’dan 1.0.1’e çıktı. Yani bir patch (yama) güncellemesi yapıldı.
- Daha sonra uygulamaya yeni bir özellik ekledin. Örneğin bir “karanlık mod” özelliği getirdin. Eski özellikler bozulmadı, her şey çalışmaya devam etti. Bu durumda sürüm numarasını 1.0.1’den 1.1.0’a yükselttin. Bu da bir minor (küçük sürüm) artışı oldu.
- Bir süre sonra kullanıcılar başka ufak bir problem daha buldu. Sen bu hatayı düzelttin ama yine yeni özellik eklemedin. Bu yüzden sürüm numarası 1.1.0’dan 1.1.1’e çıktı. Bu yine bir patch güncellemesiydi.

- Ardından çok büyük bir deęişiklik yapmaya karar verdin. Yazılımın temel mimarisini deęiřtirdin ve bazı eski fonksiyonları kaldırdın. Bu, eski kullanıcıların bazı kodlarının artık çalışmayacağı anlamına geliyordu. Bu yüzden sürüm numarasını 1.1.1’den 2.0.0’a çıkardın. Bu da bir major (ana sürüm) güncellemesi oldu.

LOCAL REPODA VERSİONLAMA

Çalışma Alanı, Commit ve Branch:

Git’te versiyonlama, dosyaları önce çalışma alanında düzenlemek, sonra staging area’ya eklemek ve commit ile kalıcı hale getirmek üzerine kuruludur. Commit, bir projenin o anki fotoğrafı gibidir. Branch ise commit zincirinin ucunu gösteren bir işaretçidir. Yeni commit geldikçe branch ucu otomatik olarak ileriye kayar. HEAD de senin şu an hangi branch veya commit üzerinde olduğunu belirtir.

Tag Kavramı:

Tag, bir commit’e sabit bir isim vermektir. Özellikle sürüm noktalarını belirtmek için kullanılır. Branch deęişebilir ama tag hep aynı commit’i gösterir. Örneğin “v1.0.0” dediğinde herkes aynı commit’i görür. Git’te tag’ler iki çeşittir: basit (lightweight) ve açıklamalı (annotated). Lightweight tag sadece isim verir, annotated tag ise tarih, yazar ve mesaj bilgisi de içerir.

Annotated Tag Oluşturmak:

Bir commit’i sürüm olarak işaretlemek için şu komut kullanılır:

```
git tag -a v1.0.0 -m "İlk kararlı sürüm"
```

Burada -a ile annotated tag oluşturulur, -m ile de açıklama eklenir. Bu şekilde oluşturulan tag, hem commit’i işaret eder hem de kimin, ne zaman ve hangi amaçla oluşturduğunu kaydeder.

Tag’leri Listelemek:

Tag’leri görmek için basitçe `git tag` yazabilirsin. Bu komut sadece isimleri listeler. Eğer açıklamaları da görmek istersen şu komut işine yarar:

```
git tag -lw
```

Bu sayede hangi sürümde hangi mesajı yazdığını da görmüş olursun.

Tag Ayrıntılarını Görmek:

Bir tag’ın hangi commit’i işaret ettiğini ve açıklamalarını öğrenmek için `git show` kullanılır:

```
git show v1.0.0
```

Bu komut, tag’ın açıklamasını, yazarını ve baęlı olduęu commit’in detaylarını gösterir.

Tag Silmek:

Yanlışlıkla oluşturduęun veya artık ihtiyaç duymadıęın bir tag’i silmek için:

```
git tag -d v1.0.0
```

Belirli Commit'e Tag Vermek:

Sadece en son commit'e değil, geçmişteki herhangi bir commit'e de tag verebilirsin. Bunun için commit hash'i yazman yeterlidir:

```
git tag v0.9.0 3e4f8d2
```

Böylece "v0.9.0" etiketi, 3e4f8d2 hash'li commit'i işaret eder.

Özet

- Annotated tag: `git tag -a vX.Y.Z -m "mesaj"`
- Tag'leri listeleme: `git tag` veya açıklamalı görmek için `git tag -lw`
- Tag detaylarını görmek: `git show vX.Y.Z`
- Tag silmek: `git tag -d vX.Y.Z`
- Belirli commit'e tag eklemek: `git tag vX.Y.Z commitHash`

AMEND

Amend Nedir ?

`git commit --amend`, son commit üzerinde değişiklik yapmanı sağlar. Git bunu eski commit'i silip yerine yeni bir commit yazarak yapar. Commit hash değiştiği için aslında tamamen yeni bir commit oluşur.

Ne İşe Yarar ?

- Commit mesajını düzeltmek
- Commit'e eklenmeyi unuttuğun dosyaları eklemek
- Fazla eklenen dosyaları çıkarmak
- Commit içeriğini güncellemek
- Push etmeden önce geçmişi "temizlemek"

Sadece Mesajı Düzeltmek:

Yanlış commit mesajı yazdın diyelim:

```
git commit --amend -m "Doğru commit mesajı"
```

Bu komut, içeriğe dokunmaz, sadece mesajı değiştirir.

Senaryo Uygulama:

1. **deneme.txt** dosyasını **oluşturdun** → Çalışma Alanı'nda (Working Directory).
2. **git add deneme.txt** yaptın → dosya Staging Area'ya geçti.
3. **git commit -m "İlk dosya eklendi"** → commit edildi ve Repository'ye kaydedildi.
4. **git log** ile baktın → geçmişte bu commit görünüyor.
5. **Başka dosyalar ekledin ve commit yaptın** → örneğin ikinci ve üçüncü commit oluştu. Artık geçmiş şöyle oldu:

```
C: Üçüncü değişiklik
B: İkinci değişiklik (mesaj veya içerik HATALI)
A: İlk dosya eklendi
```

6. **Ortakdaki commit B'yi düzeltmek istedin** → `git rebase -i A` çalıştırdın.
7. **Açılan listede pick yerine edit yazdın** → Git seni B commit'inde durdurdu.
8. **dosyada düzeltme yaptın** → mesela `deneme.txt` içine satır ekledin.
9. **git add deneme.txt** ile sahneye aldın → Staging Area'ya geçti.
10. **git commit --amend -m "İkinci değişiklik düzeltildi"** yaptın → B commit'i hem mesaj hem içerik açısından güncellendi.
11. **git rebase --continue** dedin → Git C commit'ini tekrar uyguladı ve rebase tamamlandı.
12. **git log** ile tekrar baktığında → artık geçmiş şu şekildeydi:

```
C: Üçüncü değişiklik
B': İkinci değişiklik düzeltildi (yeni hash ile)
A: İlk dosya eklendi
```


Hazırlayan: Kasım TEKE