UYGULAMALI C# BACKEND REHBERİ

LINQ ve Entity Framework Core

Temel kavramlar, sorgu yürütme mantığı, SQL'e dönüşüm ve günlük backend senaryoları.

Kimler için? C# ile backend geliştiren ve LINQ ile EF Core'u temelden sağlam biçimde öğrenmek isteyenler.

İçerik: Açıklamalı kod örnekleri, kavram karşılaştırmaları, performans notları, tek sayfalık akış şeması ve çözümlü alıştırmalar.

ANA YAKLAŞIM

Bir sorguda üç soruya cevap verebilmek gerekir: **Hangi veri isteniyor? Sorgu ne zaman çalışıyor? İşlem veritabanında mı, C# tarafında mı gerçekleşiyor?**

İçindekiler

Başlıklara tıklanarak doğrudan ilgili bölümlere geçilebilir.

GİRİŞ

Rehberi nasıl kullanmalı? 03

LINQ → EF Core → SQL: Görsel özet 04

LINQ TEMELLERİ

01 LINQ ve deklaratif düşünme 05

02 Lambda, Func ve temel LINQ operatörleri 05

03 Method syntax ve tip dönüşümü 07

04 Deferred execution ve materialization 07

05 IEnumerable, IEnumerator ve foreach 09

06 Extension method, yield return ve Where 09

ENTITY FRAMEWORK CORE

07 IQueryable, Expression Tree ve C#/SQL sınırı 11

08 Entity Framework Core ve DbContext 12

09 Temel CRUD ve SaveChangesAsync 14

10 Change Tracking ve AsNoTracking 14

11 İlişkiler, Include ve projection 15

12 Gerçek bir backend sorgusunun SQL karşılığı 16

UYGULAMA VE TEKRAR

13 Performans kontrol listesi 17

Cep özeti 18

14 Alıştırmalar ve cevap anahtarı 18

Başvuru kaynakları 19

Rehberi nasıl kullanmalı?

LINQ öğrenmenin en iyi yollarından biri, sorguyu çalıştırmadan önce hangi sonucun geleceğini tahmin etmektir. Bu kaynakta önce koleksiyonlar üzerindeki LINQ işlemleri ele alınıyor; sonra aynı yaklaşımın EF Core ile nasıl SQL sorgusuna dönüştüğü gösteriliyor.

ÖRNEKLERİN UYGULANMASI
LINQ örnekleri güncel bir .NET Console uygulamasında çalıştırılabilir. System.Linq ad alanı gerekir; modern .NET projelerinde çoğunlukla implicit using ile kullanılabilir. EF Core örnekleri için projenin .NET sürümüyle uyumlu EF Core ve veritabanı sağlayıcı paketlerini eklemek gerekir.

Okuma sırası

Kısım	İçerik	Çıkarım
A · LINQ	Koleksiyonlar, lambda, Where/Select	Sorguların nasıl yazıldığı
B · Çalışma mantığı	Deferred execution, yield, IQueryable	Sorguların ne zaman çalıştığı
C · EF Core	DbContext, CRUD, tracking, ilişkiler	Verinin nasıl okunup değiştirildiği
D · Uygulama	SQL örneği, performans, alıştırmalar	Sağlıklı sorgu yazma alışkanlığı

Çalışırken dikkat edilecek dört nokta

01 Sorgunun veri kaynağı
List<T> ile DbSet<T> üzerinde aynı LINQ ifadesi farklı yerde çalışabilir.

02 Sorgunun çalışma zamanı
Where ve Select çoğunlukla sorguyu hazırlar; ToList gibi işlemler sonucu ister.

03 SQL'e dönüştürülen işlemler
Filtreleme ve alan seçimi mümkün olduğunca veritabanında yapılmalıdır.

04 Gerekli verilerin seçimi
Salt okunur sorgularda projection ve gerektiğinde AsNoTracking tercih edilir.

LINQ → EF Core → SQL

C# ile yazılan sorgu veritabanına nasıl ulaşır?

BAŞLANGIÇ · C#

```
context.Users.Where(u => u.IsActive)
    .Select(u => new { u.Id, u.Name });
```

01

IQueryable

Sorgu oluşturulur; veriler henüz okunmaz.

Filtre ve seçilecek alanlar tanımlanır.

02

Expression Tree

Lambda ifadesinin yapısı provider'a aktarılır.

Örnek: "IsActive alanı true mu?"

03

ToListAsync() çağrısı

Sorgunun çalıştırılması istenir.

Çalıştırma isteği EF Core'a iletilir.

04

EF Core sağlayıcısı → SQL

LINQ ifadesi SQL komutuna çevrilir.

SELECT Id, Name FROM Users WHERE IsActive = 1

05

Veritabanı → sonuç

Filtre veritabanında uygulanır, satırlar döner.

Sonuç C# tarafında listeye dönüştürülür.

NERESİ VERİTABANINDA, NERESİ C# TARAFINDA?

Where / Select (IQueryable): SQL'e çevrilebilir.

ToListAsync sonrası: Sonuç artık uygulama belleğindedir.

AsEnumerable: Hemen çalıştırmaz; sonraki LINQ adımları C#'ta işlenir.

Şema temsilidir. Üretilen SQL, model ve veritabanı sağlayıcısına göre değişebilir.

01 LINQ ve deklaratif düşünme

LINQ (Language Integrated Query), C# içinde koleksiyonlar ve farklı veri kaynakları üzerinde sorgu yazmayı sağlayan dil özellikleri ve standart sorgu operatörleri bütünüdür. Aynı LINQ yaklaşımı listelerde ve EF Core'da görülebilir; ancak sorguyu çalıştıran mekanizma farklılık gösterebilir.

Aynı işi iki farklı yolla yapmak

```C#
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

// Döngü ile
var evenNumbers = new List<int>();
foreach (var number in numbers)
{
    if (number % 2 == 0)
        evenNumbers.Add(number);
}

// LINQ ile
var result = numbers.Where(x => x % 2 == 0).ToList();
```

İlk çözümde tek tek dolaşmayı, koşulu ve listeye eklemeyi biz yönetiyoruz. İkinci çözümde hangi elemanları istediğimizi belirtiyoruz. Bu yaklaşım **deklaratif sorgulama** olarak adlandırılır.

> **SORGUNUN ISKELETI**
> **Kaynak → Where → OrderBy → Select → ToList**. Where filtreleme, OrderBy sıralama, Select sonuç biçimini belirleme, ToList ise sonucu somut bir listeye dönüştürme işidir.

02 Lambda, Func ve temel LINQ operatörleri

Lambda ifadesini okumak

```C#
x => x.Age >= 18

// Yaklaşık olarak şu metoda karşılık gelir:
bool IsAdult(User user)
{
    return user.Age >= 18;
}
```

Okunuşu basit: "Bir kullanıcı al, yaşı 18 veya üzerindeyse true döndür." Lambda, yalnızca LINQ'ya özgü değildir; C#'ta kısa bir fonksiyon tanımlamanın yollarından biridir.

Func<T, TResult> nedir?

Func, parametre alan ve sonuç döndüren bir fonksiyonun türünü ifade eden delegate ailesidir.

```C#
Func<int, bool> isEven = x => x % 2 == 0;
Func<int, int> doubleIt = x => x * 2;
Func<User, string> getName = x => x.Name;
```

İfade	Girdi → çıktı	Nerede işe yarar?
x => x.Age >= 18	User → bool	Where koşulu
x => x.Name	User → string	Select dönüşümü
x => x * 2	int → int	Sayısal dönüşüm

Where ve Select

```C#
var activeAdults = users
    .Where(x => x.Age >= 18 && x.IsActive);

var names = users.Select(x => x.Name);

var cards = users.Select(x => new UserListDto
{
    Id = x.Id,
    Name = x.Name
});
```

Where, hangi elemanların kalacağını belirler ve eleman türünü değiştirmez. **Select**, her elemandan yeni bir değer üretir; bu değer string, sayı, anonim nesne veya DTO olabilir.

> **TIPLERI TAKIP ET**
> List<User> → Where → IEnumerable<User> → Select(x => x.Name) → IEnumerable<string>. Sorguyu okurken eleman tipinin nerede değiştiğini görmek çok işe yarar.

Sıralama: OrderBy ve ThenBy

```C#
var sorted = users
    .OrderBy(x => x.Country)
    .ThenBy(x => x.Name);

var latest = orders.OrderByDescending(x => x.CreatedAt);
```

Birden fazla ölçüte göre sıralamada ilk anahtarı **OrderBy** ile, sonraki anahtarları **ThenBy** ile ekle. İkinci kez OrderBy yazmak önceki sıralamayı ikincil sıralama olarak sürdürmez.

First, FirstOrDefault, Single, SingleOrDefault

Operatör	0 eşleşme	1 eşleşme	2+ eşleşme
First	Hata	İlk kayıt	İlk kayıt
FirstOrDefault	Varsayılan	İlk kayıt	İlk kayıt
Single	Hata	Tek kayıt	Hata
SingleOrDefault	Varsayılan	Tek kayıt	Hata

Varsayılan değer referans türlerinde genellikle **null** olur. Single, “bu koşulun tam olarak bir karşılığı olmalı” beklentisini kodda gösterir. Gerçek veri tekilliği için veritabanında ayrıca **unique constraint/index** gerekir.

Any, All ve toplulaştırma

```C#
bool exists = users.Any(x => x.IsActive);
bool everyoneAdult = users.All(x => x.Age >= 18);

var count = users.Count();
var total = orders.Sum(x => x.Total);
var average = users.Average(x => x.Age);
var minimum = products.Min(x => x.Price);
var maximum = products.Max(x => x.Price);
```

Any “en az bir tane var mı?”, All “hepsi uyuyor mu?” sorusunu yanıtlar. Boş bir dizide Any false, All true döndürür. Varlık sorgularında Count(...) > 0 yerine genellikle Any(...) kullanmak daha açık ve daha verimli bir niyettir.

03 Method syntax ve tip dönüşümü

LINQ, metot zinciriyle veya SQL’e benzeyen query syntax ile yazılabilir. Her iki biçim de standart sorgu operatörlerine dayanır.

```C#
// Method syntax
var names = users
    .Where(x => x.Age >= 18)
    .OrderBy(x => x.Name)
    .Select(x => x.Name);

// Query syntax
var names2 =
    from user in users
    where user.Age >= 18
    orderby user.Name
    select user.Name;
```

Backend projelerinde method syntax sık görülür. Join ve GroupBy gibi işlemlerde query syntax bilmek de başka geliştiricilerin kodunu okumayı kolaylaştırır.

> **ÜÇ KISA KURAL**
> **Where:** User → User (filtreler).
> **Select:** User → string/DTO (dönüştürür).
> **OrderBy:** Elemanın türünü değiştirmez; sıralama ölçütü ekler.

04 Deferred execution ve materialization

Birçok LINQ operatörü sorgu tanımlandığında liste üretmez. Sonuçlar, üzerlerinde dolaşıldığında veya sonuç isteyen bir operatör çağrıldığında hesaplanır. Buna **deferred execution** (ertelenmiş yürütme) denir.

```C#
var numbers = new List<int> { 1, 2, 3 };
var query = numbers.Where(x => x > 1);

numbers.Add(4);

var result = query.ToList(); // 2, 3, 4
```

4 sayısı sonuçtadır; çünkü filtre, listeye 4 eklendikten sonra çalışmıştır. Ancak önce ToList çağrılmış olsaydı eldeki sonuç listesi sonraki eklemeden etkilenmezdi.

Sorgu ne zaman çalışır?

İşlem	Genel davranış
Where, Select, Take	Çoğunlukla ertelenmiş sorgu/sequence oluşturur.
foreach	Sequence üzerinde dolaşır.
ToList, ToArray	Sonuçları somut koleksiyona dönüştürür.
Any, First	Gereken sonucu üretir; erken durabilir.
Count, Sum, Average	Hesaplamayı yapar; kaynağa göre optimize edilebilir.

Konsol çıktısıyla görmek

```C#
static bool IsEven(int x)
{
    Console.WriteLine($"Kontrol: {x}");
    return x % 2 == 0;
}

var numbers = new List<int> { 1, 2, 3, 4 };
var query = numbers.Where(IsEven);
Console.WriteLine("Sorgu hazır");

foreach (var number in query)
    Console.WriteLine($"Sonuç: {number}");
```

Önce “Sorgu hazır” yazılır. “Kontrol” satırları foreach başladığında görünür; uygun eleman bulunduğunda da “Sonuç” satırı yazılır.

Aynı sorguyu iki kez çalıştırmak

```C#
var query = context.Users.Where(x => x.IsActive);

bool exists = await query.AnyAsync();     // SQL sorgusu 1
var list = await query.ToListAsync();      // SQL sorgusu 2
```

İki ayrı sonuç talebi veritabanına iki sorgu gönderebilir. Bazen ihtiyaç budur; bazen de gereksiz trafik oluşturur. Kullanıma göre karar vermek gerekir.

ERKEN TOLIST UYARISI

EF Core'da ToListAsync filtrelerden önce çağrılırsa gereksiz kayıtlar uygulamaya taşınabilir.
Where ve Select gibi SQL'e çevrilebilen işlemler önce sorguya eklenmeli;
sonuç, bu adımlardan sonra alınmalıdır.

05 IEnumerable, IEnumerator ve foreach

IEnumerable<T>, bir koleksiyonun veya veri akışının elemanlarının sırayla gezilebildiğini gösteren arayüzdür. List<T>, diziler ve HashSet<T> gibi birçok tür bunu uygular.

```C#
IEnumerable<int> numbers = new List<int> { 1, 2, 3 };

foreach (var number in numbers)
{
    Console.WriteLine(number);
}
```

IEnumerator<T> ise sırayla gezme işleminin o anki durumunu tutar. **MoveNext()** ilerler, **Current** mevcut elemanı verir.

```C#
using var enumerator = numbers.GetEnumerator();
while (enumerator.MoveNext())
{
    Console.WriteLine(enumerator.Current);
}
```

Bu örnek foreach'in temel çalışma fikrini gösterir. foreach, uygun enumerator üzerinde ilerler ve gerekiyorsa onu dispose eder.

KISACA
IEnumerable: Gezilebilen kaynak.
IEnumerator: Gezinmenin mevcut durumu.
foreach: Kaynağı dolaşan sözdizimi.

06 Extension method, yield return ve kendi Where metodumuz

Where, List sınıfının içine yazılmış bir metot değildir. LINQ operatörlerinin çoğu **extension method** olarak sunulur. Böylece statik bir metodu sanki bir nesnenin metoduymuş gibi çağırabiliriz.

Küçük bir extension method

```C#
public static class StringExtensions
{
    public static bool IsLong(this string value)
    {
        return value.Length > 10;
    }
}

bool isLong = "Backend Developer".IsLong();
```

Buradaki **this string value**, metodun string üzerinde doğrudan çağrılmasını sağlar. Derleyici, çağrıyı temel olarak StringExtensions.IsLong(value) biçiminde ele alır.

Kendi Where metodumuz: hemen çalışan sürüm

```C#
public static IEnumerable<T> MyWhere<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate)
{
    var result = new List<T>();
    foreach (var item in source)
    {
        if (predicate(item))
            result.Add(item);
    }
    return result;
}
```

Burada MyWhere çağrılır çağrılmaz bütün kaynak dolaşılır ve yeni liste oluşturulur. Yani işlem ertelenmez.

yield return ile ertelenmiş sürüm

```C#
public static IEnumerable<T> MyWhere<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate)
{
    foreach (var item in source)
    {
        if (predicate(item))
            yield return item;
    }
}
```

Bu kez uygun elemanlar tüketici ihtiyaç duydukça üretilir. Derleyici, yield return içeren metodu kendi durumunu takip edebilen bir iterator'a dönüştürür. İki MyWhere örneği birbirinin alternatifi olduğu için aynı sınıfta aynı imzayla birlikte tanımlanmamalıdır.

İşlemler birbirine nasıl aktarılır?

```C#
var result = numbers
    .Where(x => x > 2)
    .Select(x => x * 10)
    .Take(2)
    .ToList();
```

Kaynak 1, 2, 3, 4, 5 ise sonuç 30 ve 40 olur. Where ve Select gibi akış tabanlı operatörler elemanları sırayla aktarabilir; Take iki uygun sonuçtan sonra durabilir.

> **KÜÇÜK AMA ÖNEMLI FARK**
> **Deferred execution**, her operatörün elemanları tek tek anında aktardığı anlamına gelmez. Örneğin OrderBy sıralama yapabilmek için çoğu zaman girdinin tamamını değerlendirmek zorundadır.

07 IQueryable, expression tree ve C#/SQL sınırı

Listeler üzerinde çalışan LINQ ile EF Core LINQ sorguları birbirine benzer. Fakat hangisinin nasıl çalışacağı, **sorgu sağlayıcısına (provider)** ve kullanılan metoda bağlıdır.

IEnumerable<T>	IQueryable<T>
Elemanların enumerate edilmesini sağlar.	Sorgu ifadesini bir provider'a sunabilir.
Enumerable.Where: Func<T, bool>	Queryable.Where: Expression<Func<T, bool>>
LINQ to Objects filtreyi C# tarafında yapar.	EF Core sağlayıcısı filtreyi SQL'e çevirebilir.

Delegate ile expression tree farkı

```C#
// Çalıştırılabilir fonksiyon
Func<User, bool> filter = x => x.Age >= 18;

// Analiz edilebilir ifade ağacı
Expression<Func<User, bool>> expression =
    x => x.Age >= 18;
```

Func doğrudan çağrılabilir bir fonksiyondur. Expression ise fonksiyonun nasıl kurulduğunu veri olarak temsil eder. Örneğin x.Age >= 18 ifadesinde "Age alanı", "büyüktür/eşittir işlemi" ve "18 sabiti" ayrı düğümler olarak bulunabilir.

KAVRAMSAL GÖSTERIM

```
Lambda: x => x.Age >= 18

GreaterThanOrEqual
├── MemberAccess: Age
└── Constant: 18

Olası SQL: WHERE Age >= 18
```

EF Core ile sorgu oluşturmak

```C#
IQueryable<User> query = context.Users;
query = query.Where(x => x.IsActive);
query = query.Where(x => x.Age >= 18);

var adults = await query.ToListAsync();
```

Where çağrıları sorgu tanımını genişletir. Veritabanı sorgusu, sonuç istendiği anda - burada ToListAsync sırasında - çalıştırılır. **IQueryable her zaman SQL demek değildir**; onun nasıl yorumlanacağı provider'a bağlıdır.

AsEnumerable ne yapar?

```C#
var query = context.Users
    .Where(x => x.IsActive)     // SQL'e çevrilebilir
    .AsEnumerable()
    .Where(x => IsSpecialUser(x)); // C# tarafında
```

AsEnumerable tek başına sorguyu o anda çalıştırmaz. Fakat sonraki LINQ çağrılarının Enumerable sürümlerine yönelmesine neden olur. Böylece sonraki filtreler, veri uygulamaya gelirken C# tarafında çalışır.

Her C# fonksiyonu SQL'e çevrilemez

```C#
var users = await context.Users
    .Where(x => IsSpecialUser(x))
    .ToListAsync();
```

EF Core, kullanıcı tarafından yazılmış keyfî bir metodun davranışını SQL olarak bilemeyebilir; bu durumda sorgu hata verebilir. Gerekirse SQL'e çevrilebilen ifadeler kurmak veya sınırlı veri üzerinde istemci tarafı işlem yapmak gerekir.

> **PRATIK KARŞILAŞTIRMA**
> **Riskli:** context.Users.ToListAsync() ile önce bütün kullanıcıları alıp sonra Where çalıştırmak.
> **Genellikle daha iyi:** context.Users.Where(...).Select(...).ToListAsync() ile süzme ve seçmeyi veritabanına bırakmak.

08 Entity Framework Core'un temel modeli

Entity Framework Core, C# nesneleri ile ilişkisel veritabanı tabloları arasında eşleme kuran bir ORM'dir (Object-Relational Mapper). LINQ sorgularını SQL'e çevirebilir; kayıt ekleme, güncelleme ve silme işlemlerini yönetir.

```C#
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
    public int Age { get; set; }
    public bool IsActive { get; set; }
    public string Country { get; set; } = string.Empty;
}
```

C# tarafı	Veritabanı tarafındaki karşılığı
User sınıfı	Örneğin Users tablosu
User nesnesi	Bir satır
Property	Kolon
Foreign key	Başka tablonun anahtarına bağlanan alan

DbContext ve DbSet

```C#
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }

    public DbSet<User> Users => Set<User>();
    public DbSet<Order> Orders => Set<Order>();
}
```

DbContext, sorgu yazma, değişiklikleri takip etme ve veritabanına kaydetme işlerinin merkezidir. Genellikle kısa ömürlü kullanılır. **DbSet<T>** ise ilgili entity türüne erişmek için başlangıç noktasıdır.

> **ORM VE SQL İLİŞKİSİ**
> EF Core, SQL'i ortadan kaldırmaz; uygulamada yazılan LINQ ifadeleri ve model tanımlarından SQL üretir. Bu yüzden sorgunun veritabanında ne yaptığını düşünmek önemlidir.

09 Temel CRUD ve SaveChangesAsync

Create — yeni kayıt eklemek

```C#
var user = new User
{
    Name = "Ali",
    Email = "ali@example.com",
    Age = 25,
    IsActive = true
};

context.Users.Add(user);
await context.SaveChangesAsync();
```

Read — kayıt okumak

```C#
var user = await context.Users
    .SingleOrDefaultAsync(x => x.Id == id);
```

Update — takip edilen kaydı değiştirmek

```C#
var user = await context.Users.SingleAsync(x => x.Id == id);
user.Name = "Mehmet";
await context.SaveChangesAsync();
```

Delete — kayıt silmek

```C#
var user = await context.Users.SingleAsync(x => x.Id == id);
context.Users.Remove(user);
await context.SaveChangesAsync();
```

Ekleme, değiştirme ve silme işlemleri DbContext tarafından takip edilir. Çoğu durumda veritabanına yazma işlemi **SaveChangesAsync** çağrısında gerçekleşir. Gerçek bir API'de "kayıt bulunamadı", doğrulama, benzersiz alanlar ve hata yönetimi ayrıca ele alınmalıdır.

10 Change Tracking ve AsNoTracking

EF Core, varsayılan tracking sorgularında okunan entity'leri takip eder. Bir property değiştiğinde ve SaveChanges çağrıldığında gerekli güncelleme yapılabilir.

```C#
var user = await context.Users.SingleAsync(x => x.Id == id);
// Veritabanındaki ad: "Ali"

user.Name = "Ahmet";
await context.SaveChangesAsync();
```

Burada ayrıca Update(user) çağırmaya gerek yoktur; aynı DbContext tarafından izlenen entity zaten değiştirilmiştir.

Salt okunur sorgular için AsNoTracking

```C#
var users = await context.Users
    .AsNoTracking()
    .Where(x => x.IsActive)
    .Select(x => new UserListDto
    {
        Id = x.Id,
        Name = x.Name
    })
    .ToListAsync();
```

Tracking	No-tracking
Entity değişikliklerini takip eder.	Entity Change Tracker'a alınmaz.
Güncelleme akışlarında kullanışlıdır.	Salt okuma akışlarında uygundur.
Takip maliyeti vardır.	Takip maliyeti azaltılabilir.

Yalnızca basit alanlardan DTO oluşturan projection'larda zaten takip edilecek entity bulunmayabilir. Bu nedenle AsNoTracking'in ek etkisi bu tip sorgularda sınırlı olabilir.

11 İlişkiler, Include ve projection

İlişkisel veritabanlarında tablolar foreign key üzerinden birbirine bağlanır. EF Core'da **navigation property**, ilişkili nesneye veya nesne koleksiyonuna erişmek için kullanılır.

```C#
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public bool IsActive { get; set; }
    public ICollection<Order> Orders { get; set; }
        = new List<Order>();
}

public class Order
{
    public int Id { get; set; }
    public decimal Total { get; set; }
    public int CustomerId { get; set; } // Foreign key
    public Customer Customer { get; set; } = null!;
    public DateTime CreatedAt { get; set; }
    public OrderStatus Status { get; set; }
}

public enum OrderStatus { Pending, Completed }
```

Bu ilişki **one-to-many** türündedir: Bir müşterinin birden fazla siparişi bulunabilir; her siparişin bir müşterisi vardır.

DTO örneklerinde kullanılan tipler

```C#
public class UserListDto
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
}

public class OrderDto
{
    public int Id { get; set; }
    public string CustomerName { get; set; } = string.Empty;
    public decimal Total { get; set; }
}
```

Include ile ilişkili entity'yi yüklemek

```C#
var orders = await context.Orders
    .Include(x => x.Customer)
    .ToListAsync();

// Customer üzerinde Address navigation'ı varsa:
var details = await context.Orders
    .Include(x => x.Customer)
        .ThenInclude(x => x.Address)
    .ToListAsync();
```

DTO için Select kullanmak

```C#
var orders = await context.Orders
    .AsNoTracking()
    .Select(x => new OrderDto
    {
        Id = x.Id,
        CustomerName = x.Customer.Name,
        Total = x.Total
    })
    .ToListAsync();
```

Burada müşteri adını almak için EF Core gerekli JOIN'i üretebilir. API yanıtında yalnızca belirli alanlar gerekiyorsa bütün ilişkili nesneyi Include ile yüklemek yerine Select kullanmak daha az veri çekebilir.

> **INCLUDE MI, SELECT MI?**
> **Include:** İlişkili entity'lerle birlikte çalışman gerekiyorsa.
> **Select:** Ekrana, API yanıtına veya rapora yalnızca bazı alanlar dönecekse.

12 Gerçek bir backend sorgusunun SQL karşılığı

Örnek ihtiyaç: Aktif müşterilerin son 30 gündeki tamamlanmış siparişlerini, tutarı yüksekten düşüğe sırala. İlk 20 siparişin ID, müşteri adı ve toplam tutarını getir.

```C#
var cutoffDate = DateTime.UtcNow.AddDays(-30);

var result = await context.Orders
    .AsNoTracking()
    .Where(x => x.Customer.IsActive &&
                x.Status == OrderStatus.Completed &&
                x.CreatedAt >= cutoffDate)
    .OrderByDescending(x => x.Total)
    .Select(x => new OrderDto
    {
        Id = x.Id,
        CustomerName = x.Customer.Name,
        Total = x.Total
    })
    .Take(20)
    .ToListAsync();
```

Sorgu önce filtreleri belirliyor, sonra sıralıyor, DTO için gereken alanları seçiyor ve 20 kayıtla sınırlandırıyor. Son olarak ToListAsync çağrısıyla veritabanından sonuç alınıyor.

Yaklaşık SQL karşılığı (SQL Server)

```SQL
SELECT TOP (20)
    o.Id, c.Name AS CustomerName, o.Total
FROM Orders AS o
INNER JOIN Customers AS c ON c.Id = o.CustomerId
WHERE c.IsActive = 1
  AND o.Status = @completedStatus
  AND o.CreatedAt >= @cutoffDate
ORDER BY o.Total DESC;
```

Bu SQL örneği temsilîdir. Gerçek SQL; EF Core sürümüne, model eşlemesine ve veritabanı sağlayıcısına göre değişebilir. Gerçek SQL, ToQueryString() veya EF Core loglarıyla incelenebilir.

> **SORGUYA SQL AÇISINDAN BAKIŞ**
> WHERE neyi filtreliyor? JOIN nereden doğuyor? SELECT hangi kolonları getiriyor? ORDER BY ve TOP/LIMIT kaç kaydı etkiliyor? Bu dört soruya cevap verebilmek, EF Core performansını anlamanın iyi bir başlangıcıdır.

13 Performans kontrol listesi

Her sorguda bütün maddeleri tek tek kontrol etmek gerekmez. Özellikle yavaş çalışan veya beklenenden çok veri taşıyan sorgularda bu listeye dönmek yararlıdır.

- Sorgu hâlâ IQueryable mı, yoksa veri ToList ile belleğe alınmış mı?
- Where koşulları veritabanında mı uygulanıyor?
- Yalnızca gerekli kolonlar mı seçiliyor? DTO projection yeterli mi?
- Salt okunur sorgularda Change Tracker gerekli mi?
- Include gerekli mi? Gereksiz ilişkili veri getiriliyor mu?
- Aynı sorgu Any ve ToList gibi işlemlerle tekrar mı çalıştırılıyor?
- OrderBy/ThenBy sıralaması uygun mu? Skip/Take için kararlı bir sıralama var mı?
- Veri sayısı arttığında performans nasıl etkilenir? Uygun indeksler var mı?

Cep özeti

Kavram	Tek cümleyle
Where	Kayıtları filtreler.
Select	Kayıtları başka biçime dönüştürür.
Deferred execution	Sorgunun hazırlanması, çalıştırılması değildir.
IEnumerable	Kaynağın elemanları dolaşılabilir.
IQueryable	Sorgu sağlayıcısının yorumlayacağı ifade sunabilir.
Expression Tree	C# ifadesinin analiz edilebilir yapısıdır.
AsEnumerable	Sonraki LINQ işlemlerini C# tarafına yönlendirir.
ToListAsync	Sorguyu çalıştırır, sonucu listeye dönüştürür.
AsNoTracking	Entity takibini kapatır.
Include	İlişkili entity verisini yüklemek içindir.
Projection	Yalnızca gereken alanları seçmeye yarar.

14 Alıştırmalar ve cevap anahtarı

Soruların cevap anahtarına bakılmadan çözülmesi önerilir. Son sorudaki programın bir Console uygulamasında çalıştırılması özellikle yararlıdır.

- **1.** Where ile Select arasındaki en temel fark nedir?
- **2.** FirstOrDefault ve SingleOrDefault hangi durumda farklı davranır?
- **3.** Sorgu tanımlandıktan sonra listeye eklenen eleman sonuca girebilir mi?
- **4.** Func<User, bool> ile Expression<Func<User, bool>> arasında ne fark vardır?
- **5.** AsEnumerable() tek başına sorguyu çalıştırır mı?
- **6.** AsNoTracking neden kullanılır?
- **7.** Yalnızca ID ve isim dönülecekse Select neden uygun olabilir?
- **8.** Aşağıdaki kod hangi sırayla yazdırır?

```C#
var numbers = new List<int> { 1, 2, 3, 4, 5 };
var query = numbers
    .Where(x => {
        Console.WriteLine($"W:{x}");
        return x % 2 == 0;
    })
    .Select(x => {
        Console.WriteLine($"S:{x}");
        return x * 10;
    });

Console.WriteLine("A");
numbers.Add(6);
Console.WriteLine("B");
foreach (var x in query)
    Console.WriteLine($"R:{x}");
```

Kısa cevaplar

- **1.** Where filtreler; Select elemanları başka bir değer veya tipe dönüştürür.
- **2.** Birden fazla eşleşmede FirstOrDefault ilk kaydı döndürür; SingleOrDefault hata verir.
- **3.** Evet; deferred sorgu liste değiştikten sonra enumerate ediliyorsa yeni elemanı görebilir.
- **4.** Func çalıştırılabilir fonksiyon, Expression ise analiz edilebilir ifade ağacıdır.
- **5.** Hayır. Sonraki LINQ adımlarının Enumerable sürümlerini kullanmasına yol açar.
- **6.** Salt okuma senaryolarında gereksiz entity takip maliyetini azaltabilir.
- **7.** İstenen kolonları sınırlamak ve API yanıtının biçimini belirlemek için.

8. sorunun konsol çıktısı

```KONSOL
A
B
W:1
W:2
S:2
R:20
W:3
W:4
S:4
R:40
W:5
W:6
S:6
R:60
```

İlk iki satır sorgu enumerate edilmeden yazılır. foreach başladığında her sayı Where'den geçer; koşulu sağlayanlar hemen Select'e aktarılır. İşlem "önce tüm Where, ardından tüm Select" biçiminde ilerlemez.

Bilgiyi derinleştirmek için sonraki konular

Temel bilgilerin ardından LINQ'da SelectMany, GroupBy, Join ve Aggregate; EF Core'da ise N+1, sayfalama, indeksleme, migration, transaction ve concurrency konularıyla devam edilebilir.

Başvuru kaynakları

LINQ - https://learn.microsoft.com/dotnet/csharp/linq/
Entity Framework Core - https://learn.microsoft.com/ef/core/
EF Core ile sorgulama - https://learn.microsoft.com/ef/core/querying/
Tracking / No-tracking - https://learn.microsoft.com/ef/core/querying/tracking/