Repository Katmanı'nda Bilgi Şehrinin Sırlarını Keşfetmek

Yıllar önce, insanların sadece rüyalarında dolaşan bir dünya vardı. Bilgi kısıtlıydı, veriler elle tutulur birer hazine gibi saklanırdı. Ancak, zamanla teknolojinin ilerlemesiyle birlikte, bu dünya, görünmeyen bir ağ tarafından örülmüş devasa bir şehre dönüştü. Bu şehir, bilgilerin, anılrın ve geçmişin en saf haliyle depolandığı, korunduğu ve paylaşıldığı bir yerdi. İşte bu şehir, "**Veriopolis**" adını taşıyordu.

Veriopolis'in en değerli bölgesi, şehrin kalbi sayılan **Repository Katmanı**'ydı. Bu katman, projelerin belleği olarak adlandırılır ve Veriopolis'in geçmişini, anını, geleceğini barındıran bir arşiv gibi işlev görürdü. Ancak, bu kıymetli bellek, sadece Veriopolis'in derinliklerinde yer alan ve "**Repository**" olarak bilinen gizemli bir bölge tarafından yönetilmekteydi.

Repository, Veri Katmanı'nın bekçisi ve koruyucusu olarak görev yapıyordu. Bu bölge, şehirdeki altyapı ve enerji sistemleri gibi görünmeyen, ancak hayati öneme sahip bir rol üstleniyordu. Zamanla büyüyen **Veriopolis'in** karmaşık dokusunu düzenli tutmak, bilgileri güvenle depolamak ve gerektiğinde ulaşmak için **Repository**, şehrin gizemli kulesinde bulunan uzman bir ekip tarafından yönetiliyordu.

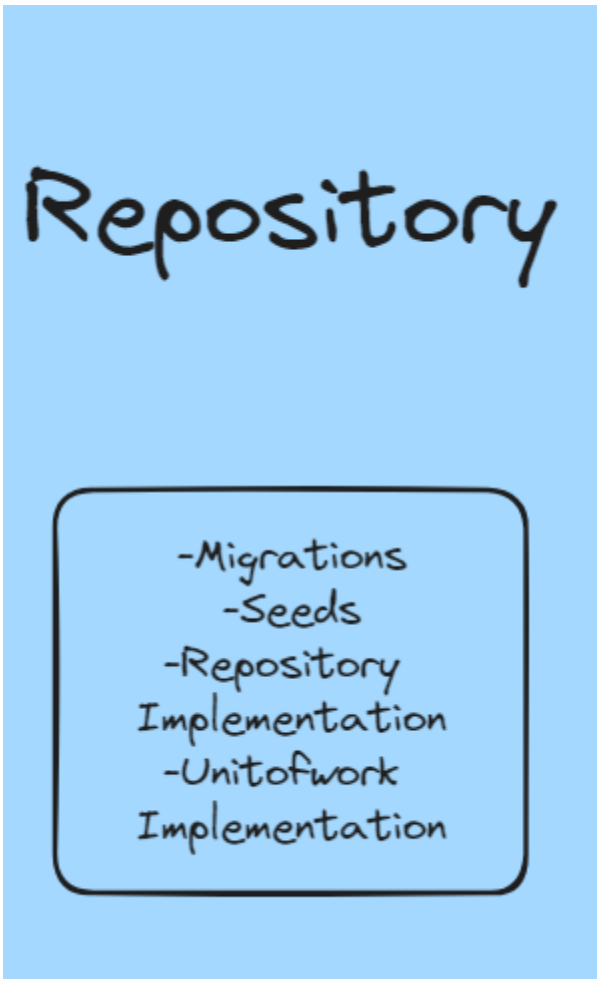
Bir gün, Veriopolis'teki bir grup araştırmacı, Repository'nin kapılarına ulaşmaya karar verdi. Bu cesur maceracılar, Veri Katmanı'ndaki bilgileri daha derinlemesine incelemek, geçmişi keşfetmek ve geleceği şekillendirmek istiyorlardı. Repository'nin kapıları, onları bilginin sonsuz koridorlarına doğru bir yolculuğa davet ediyordu.

Ancak, bu maceranın başlangıcı, sadece kapıları aralamak değil, aynı zamanda Repository'nin gizemli bekçileriyle iletişim kurmaktı. Kapının ardında nelerin beklediğini öğrenmek, Veriopolis'in sırlarını çözmek ve bilgi şehrinin derinliklerinde yeni bir sayfa açmak için cesaret gerektiriyordu.

İşte bu noktada, Veriopolis'in kalbinde başlayan bu heyecan verici hikaye, Repository'nin kapılarını aralayan cesur araştırmacıların gözünden bir keşif yolculuğuna dönüşüyordu. Bu keşif, Veri Katmanı'ndaki bilgilerin nasıl düzenlendiğini, korunduğunu ve paylaşıldığını ortaya çıkararak, Veriopolis'in sırlarını aydınlatmak için yeni bir başlangıç olacaktı.

Hazır mısınız? Veriopolis'in ve Repository Katmanı inşa etmenin sırlarını hepbirlikte keşfedelim!

Aşağıdaki oluşturduğumuz mimariye istinaden aşama aşama bu alanların neden ve niçin oluşturulduğunun detaylarını hepbirlikte göreceğiz!



AppDbContext

SQL Server'da kullanılacak bir veritabanını temsil edecek bir sınıf oluşturmak gerekiyor. Bu sınıf, genellikle **"AppDbContext"** olarak adlandırılır ve Entity Framework Core (EF Core) kütüphanesi içindeki **DbContext** sınıfından türetilir. EF Core, uygulama ile veritabanı arasında bir bağlantı kurma işlevi gören bir **ORM (Object-Relational Mapping)** aracıdır. Farklı ilişkisel veritabanlarıyla (MSSQL, PostgreSQL, MySQL, vb.) çalışabilir.

İlgili adımları gerçekleştirmek için EF Core ile ilgili kütüphaneleri yüklemeniz gerekmektedir, örneğin;

- **"Microsoft.EntityFrameworkCore".**
- Bu örnekte **MSSQL** kullanılacağı için "**Microsoft.EntityFrameworkCore.SQLServer"** kütüphanesini de yüklemek gerekmektedir.

- **“Microsoft.EntityFrameworkCore.Design”** kütüphanesini de yükleyerek tasarım zamanında **migration** işlemlerini gerçekleştirebilmek için gerekli araçları sağlamış olacaksınız.
- Migration komutlarını yazabilmek için **Visual Studio** üzerindeki **Package Manager Console**’u kullanabilirsiniz. Bu amaçla **“Microsoft.EntityFrameworkCore.Tools”** kütüphanesini indirmeniz gerekmektedir. Eğer bu kütüphaneyi indirmezseniz, migration komutlarınızı dotnet komutları kullanarak bir komut isteminden oluşturabilirsiniz.

AppDbContext sınıfı oluşturulacak ve **DbContext** sınıfından türetilerek bir constructor içerecektir. **Constructor** içinde **DbContextOptions** almanızın nedeni, bu options ile birlikte veritabanı yolunu **Program.cs** dosyasından geçirmenizdir. Bu parametreyi kolaylıkla geçirebilmek için constructor içinde alıp ardından base sınıfın options parametresine ileteceksiniz. Her bir varlık (entity) için bir **DbSet** oluşturmanız gerekmektedir. Örneğin, “**Team**”, “**User**” ve “**UserProfile**” için ayrı ayrı DbSetler oluşturmanız gerekmektedir.

```
using Microsoft.EntityFrameworkCore;

namespace YourNamespace // Kullanılacak olan namespace'i belirtiniz
{
  public class AppDbContext : DbContext
  {
    public DbSet<Team> Teams { get; set; }
    public DbSet<User> Users { get; set; }
    public DbSet<UserProfile> UserProfiles { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
}
```

Entitylerin Konfigürasyonları

Entity sınıfları içinde farklı özellikler bulundururuz. Eğer bu özelliklere özel bir düzenleme yapmazsak, özellik alanları rastgele bir şekilde veri tabanına kaydedilebilir. Veri tabanında

gereksiz yer tutmamak için bu ayarlamaları **Entity Framework (EF) Core** üzerinde yapabiliriz. Özellikle, **Primary Key** veya **Foreign Key** olarak tanımlanan alanları belirli bir düzene oturtmak önemlidir. Ancak, EF Core'a uygun formatta verildiyse zaten **Foreign Key** ve **Primary Key** zaten çalışacaktır çünkü EF Core'un standartlarına uygun isimlendirmeler yapılmıştır. Eğer bu isimlendirmeler farklı bir şekilde yapıldıysa buna uygun bir şekilde konfigürasyon yapılması şarttır.

Entitylerle ilgili ayarları yapmak için, migration işlemi sırasında **override** etmemiz gereken bir metodumuz vardır: **OnModelCreating** (*model oluştururken çalıştırılacak metod*). Geçtiğimiz yazıdan hatırlayacağımız gibi eğer isimlendirmeyi EF Core'a uygun yapmazsak attribute kullanımıyla Primary Key veya Foreign Key olarak tanımlama yapabileceğimizi belirtmiştik. Ancak gün sonunda baktığımızda kodumuzun daha kaliteli olabilmesi için entityleri mümkün olduğunca temiz tutmak önemlidir.O nedenle konfigürasyonların **OnModelCreating** içerisinde yapılması daha doğru olacaktır. **[Key]** özniteliğini **Entity** üzerinde tanımlamak doğru bir yaklaşım değildir.

*Ancak **AppDbContext** içinde bu işlemi gerçekleştirirken, unutulmaması gereken bir nokta şudur ki proje kapsamında birçok entity olabilir. Bu nedenle her işlemi burada yapmak da doğru bir yaklaşım olmayabilir. Aşağıdaki örnek doğru bir yaklaşım değildir.*

```
using Microsoft.EntityFrameworkCore;

namespace YourNamespace // Kullanılacak olan namespace'i belirtiniz
{
  public class AppDbContext : DbContext
  {
    public DbSet<Team> Teams { get; set; }
    public DbSet<User> Users { get; set; }
    public DbSet<UserProfile> UserProfiles { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    // Diğer DbContext konfigürasyonları buraya eklenebilir.

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      // Tablolar arasındaki ilişkileri belirtmek için gerekirse bu bölümü kullanabilirsiniz.
```

```
        // Örnek: Bir kullanıcının bir kullanıcı profili olması
        modelBuilder.Entity<User>()
            .HasOne(u => u.UserProfile)
            .WithOne(up => up.User)
            .HasForeignKey<UserProfile>(up => up.UserId);

        // Örnek: Bir takımın birden fazla kullanıcıya sahip olması
        modelBuilder.Entity<Team>()
            .HasMany(t => t.Users)
            .WithOne(u => u.Team)
            .HasForeignKey(u => u.TeamId);
    }
  }
}
```

Bunun için ayrı bir klasör açıp **Configuration** işlemlerini bu klasör içinde yapmak daha uygundur. Örneğin, "**TeamConfiguration**" adlı bir class oluşturabiliriz. Bu Configuration class'ını EF Core ile gelen "**IEntityTypeConfiguration**" dosyasından miras alması gerekir. Bu miras içerisinde implementasyonu çalıştırınca bu interfaceden "**Configure**" adında bir metodun gelmesi beklenir. İşte bu metodun içerisinde, **Team** ile ilgili tüm konfigürasyonları yapabiliriz.

*Örneğin, **Primary Key**'in nasıl belirleneceğini veya **Identity** olarak nasıl artırılacağını belirtebiliriz. **UseIdentityColumn**'a hiçbir şey vermezsek, otomatik olarak birer birer artacaktır.*

```
// Primary Key tanımlama ve otomatik artış kullanma
    builder.Property(t => t.Id)
        .UseIdentityColumn(); // Otomatik artış için
```

*Eğer bir alanın zorunlu ve nullable olmamasını istiyorsak "**require**" diyebiliriz.*

```
    builder.Property(t => t.TeamName)
        .HasMaxLength(50)
        .IsRequired();
```

*Tablonun ismini belirtmek istersek "**toTable**" fonksiyonunu kullanırız. Normal şartlarda bir şey belirtmezsek, **AppDbContext**'te DbSet üzerinde verdiğimiz ismi alır. ForeignKey'i eğer standartlara göre yazmazsak, ilgili konfigürasyonu yapmamız gereklidir.*

```
// Tablo adı belirleme (Opsiyonel)
    builder.ToTable("Teams");
```

Team Configuration

TeamConfiguration sınıfı, **Team** entity tipi için **Fluent API** kullanarak veritabanı modeli için özel ayarları tanımlar.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class TeamConfiguration : IEntityTypeConfiguration<Team>
{
  public void Configure(EntityTypeBuilder<Team> builder)
  {
    //Fluent API Ayarlamaları
    builder.HasKey(x => x.Id);

    // Primary Key tanımlama ve otomatik artış kullanma
    builder.Property(t => t.Id)
        .UseIdentityColumn(); // Otomatik artış için

    // TeamName alanı için maksimum uzunluğu belirleme ve zorunlu hale getirme
    builder.Property(t => t.TeamName)
        .HasMaxLength(100)
        .IsRequired();


    // Tablo adı belirleme (Opsiyonel)
    builder.ToTable("Teams");
  }
```

User Configuration

UserConfiguration sınıfı, **User** entity tipi için **Fluent API** kullanarak veritabanı modeli için özel ayarları tanımlar.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace YourNamespace // Kullanılacak olan namespace'i belirtiniz
```

```
{
  public class UserConfiguration : IEntityTypeConfiguration<User>
  {
    public void Configure(EntityTypeBuilder<User> builder)
    {
      //Fluent API Ayarlamaları
      builder.ToTable("Users"); // Tablo adını belirtiniz

      builder.HasKey(u => u.Id); // Primary key belirleme ( Uygun formattayse gerek yok.)
      builder.Property(t => t.Id)
        .UseIdentityColumn(); // Otomatik artış için


    }
  }
}
```

UserProfile Configuration

UserProfileConfiguration sınıfı, **UserProfile** entity tipi için **Fluent API** kullanarak veritabanı modeli için özel ayarları tanımlar.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace YourNamespace // Kullanılacak olan namespace'i belirtiniz
{
  public class UserProfileConfiguration : IEntityTypeConfiguration<UserProfile>
  {
    public void Configure(EntityTypeBuilder<UserProfile> builder)
    {
      //Fluent API Ayarlamaları
      builder.ToTable("UserProfile"); // Tablo adını belirtiniz

      builder.HasKey(up => up.Id); // Primary key belirleme( Uygun formattayse gerek yok.)
      builder.Property(t => t.Id)
        .UseIdentityColumn(); // Otomatik artış için

    }
```

```
    }
}
```

Fluent API ve AppDbContext.cs üzerinde konfigurasyonların tanımlanması

Fluent API, bir API tasarım tarzıdır ve bu tasarım tarzını kullanarak bir dilin akıcı (fluid) ve okunabilir bir şekilde oluşturulmasına odaklanır. Entity Framework (EF) Core'da Fluent API, veritabanı modelini kod içinde tanımlamak için kullanılan bir yaklaşımdır. Bu API, geleneksel olarak yapılandırma dosyaları veya öznitelik tabanlı yapılandırmadan daha esnek ve geniş bir kontrol sunar.

Fluent API, modeli tanımlamak ve **ORM** (Object-Relational Mapping) kütüphanesi olan EF Core'un nasıl davranacağını özelleştirmek için zincirleme metot çağrıları şeklinde kullanılır. Bu, kodun daha açık, okunabilir ve özelleştirilebilir olmasına olanak tanır. Ayrıca, Fluent API kullanarak veritabanı modelini kod içinde daha ayrıntılı bir şekilde kontrol etme imkanı sağlar.

Fluent API kullanımının özeti, **AppDbContext** sınıfında **OnModelCreating** metodunda model konfigürasyonlarını ayarlamak için EF Core tarafından sağlanan bir yöntemdir. Bu yöntemle, model özellikleri, ilişkiler ve diğer ayarlar Fluent API aracılığıyla belirlenir. Bu konfigürasyonları tek tek belirtmek yerine, genellikle bir dizi ayrı sınıf içinde yapılır ve **OnModelCreating** metodunda bu sınıflar **ApplyConfigurationFromAssembly** metodu ile uygulanır.

Aşağıdaki kod satırı **Entity Framework Core** tarafından sağlanan **Fluent API** konfigürasyonlarını uygulamanızı kolaylaştıran bir yöntemi temsil eder. Bu satır, aynı assembly içinde yer alan ve **IEntityTypeConfiguration<TEntity>** arayüzünü uygulayan tüm Fluent API konfigürasyon sınıflarını otomatik olarak tanımlar ve uygular. Kısacası, bu satır, mevcut assembly içindeki tüm **Fluent API** konfigürasyon sınıflarını toplar ve bunları model yapılandırması için kullanır. Bu sayede, her bir entity tipi için özel konfigürasyonları tek tek belirtmek yerine, bu sınıfları tek bir yerden toplu bir şekilde tanımlayabilir ve uygulayabilirsiniz. Bu, kodun daha düzenli ve bakımı daha kolay hale gelmesini sağlar.

```
modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
```

Aşağıda **AppDbContext.cs** üzerindeki uygulanmış halini görebilirsiniz.

```
using System.Reflection;

namespace YourNamespace // Kullanılacak olan namespace'i belirtiniz
{
```

```
  public class AppDbContext : DbContext
  {
    public DbSet<Team> Teams { get; set; }
    public DbSet<User> Users { get; set; }
    public DbSet<UserProfile> UserProfiles { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);

      // Assembly içinde bulunan tüm Fluent API konfigürasyon sınıflarını uygula

modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
    }
  }
}
```

*Eğer **AppDbContext.cs** üzerinde konfigurasyon dosyalarını **Assembly** üzerinden çekmek yerine tek tek tanımlamak isterseniz aşağıdaki şekilde kullanım gerçekleştirmeniz gerekmektedir. Ama şunu unutmayın ki her konfigürasyon oluşturduğunuzda buraya ekleme yapmanız gerekecektir. Bu da çok sağlıklı bir yaklaşım olmayabilir.*

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.ApplyConfiguration(new TeamConfiguration());
  modelBuilder.ApplyConfiguration(new UserConfiguration());
  modelBuilder.ApplyConfiguration(new UserProfileConfiguration());

  // Diğer özel konfigürasyonlar buraya eklenebilir.
}
```

SeedData

SeedData, bir uygulamanın başlangıcında veya belirli bir noktada veritabanına varsayılan (sabit) verileri eklemek veya başlangıç durumunu tanımlamak için kullanılan verilerdir.

Genellikle uygulamanın ilk kez çalıştırılması veya bir güncelleme sonrasında veritabanındaki başlangıç durumunu belirlemek amacıyla kullanılır.

SeedData, genellikle sabit veya ön tanımlı verileri temsil eder ve bu veriler uygulamanın çalışması için gereken başlangıç bilgilerini içerebilir. Örneğin, bir blog uygulaması için varsayılan kategoriler, kullanıcı rolleri veya örnek makaleler seed data olabilir. Bu veriler, uygulamanın başlatılmasından önce veritabanına eklenir ve uygulamanın ilk kullanımı sırasında kullanıcılara önceden tanımlanmış bir başlangıç durumu sunar.

SeedData, uygulamanın ilk kurulumu veya geliştirme aşamasında kullanışlıdır ve veritabanındaki başlangıç durumunu belirlemek için kullanıcıya kolaylık sağlar.

SeedData'nın Projelerde Uygulanması

Veritabanında ilgili tabloları oluştururken ve kayıtlara varsayılan değerler atarken, bu süreç iki aşamada gerçekleşebilir:

- Migration işlemi sırasında ilgili tablolar oluşturulurken varsayılan verilerin atanması,
- Migration oluşturulduktan ve tablolar oluşturulduktan sonra uygulama başladığında verilerin eklenmesi.

Bu işlemi gerçekleştirmek için **"Seeds"** adında bir klasör oluşturmak gereklidir. Bu klasör içinde varsayılan veriler tanımlanır, ve bu tanımlamalar, **Configuration** mantığıyla uyumlu olmalıdır. Seed işlemini **AppDbContext** içinde de gerçekleştirebilirim, ancak bu durumda kod karmaşıklığına yol açılabilir. **Seed** işlemi sırasında tabloda varsayılan kayıtları oluşturmak istiyorsak, Id'leri manuel olarak atmak gerekir. Normalde, Id'ler otomatik artan bir şekilde atanır, ancak Seed Data sırasında bu değerleri kendiniz belirtmeniz gerekmektedir.

Team Seed

```
public class TeamSeed : IEntityTypeConfiguration<Team>
{
  public void Configure(EntityTypeBuilder<Team> builder)
  {
    builder.HasData(
      new Team { Id = 1, TeamName = "Development" },
      new Team { Id = 2, TeamName = "Marketing" },
      new Team { Id = 3, TeamName = "Sales" }
      // Diğer seed verileri buraya eklenebilir.
    );
```

```
    }
}
```

User Seed

```
public class UserSeed : IEntityTypeConfiguration<User>
{
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.HasData(
            new User { Id = 1, UserName = "john_doe", Email = "john@example.com", Password =
"password123", TeamId = 1 },
            new User { Id = 2, UserName = "jane_doe", Email = "jane@example.com", Password =
"password456", TeamId = 2 },
            new User { Id = 3, UserName = "bob_smith", Email = "bob@example.com", Password
= "password789", TeamId = 1 }
            // Diğer seed verileri buraya eklenebilir.
        );
    }
}
```

UserProfile Seed

```
public class UserProfileSeedConfiguration : IEntityTypeConfiguration<UserProfile>
{
    public void Configure(EntityTypeBuilder<UserProfile> builder)
    {
        builder.HasData(
            new UserProfile { Id = 1, FirstName = "John", LastName = "Doe", NickName = "JD",
UserId = 1 },
            new UserProfile { Id = 2, FirstName = "Jane", LastName = "Doe", NickName = "Jane",
UserId = 2 },
            new UserProfile { Id = 3, FirstName = "Bob", LastName = "Smith", NickName = "Bob",
UserId = 3 }
            // Diğer seed verileri buraya eklenebilir.
        );
    }
}
```

GenericRepository

Core katmanında oluşturduğum **IGenericRepository** arabirimini, **Repository** katmanında uygulayacağım. Bu amaçla bir "**Repositories**" klasörü oluşturup içine **GenericRepository** sınıfını eklemem gerekiyor. Bu sınıf tüm entity'ler için genel geçer olacak şekilde tasarlanacaktır.

AppDbContext property'sini protected olarak tanımlamam mantıklı olacaktır çünkü , sadece bir entitye ait temel CRUD metodları dışında özel metodlara ihtiyaç duyulması durumunda özelleştirilmiş Repository'ler oluşturulacaktır. Bu durumda, AppDbContext'e ihtiyaç olabilir, bu nedenle onu protected olarak belirledik. Böylece bu metodun sadece miras alan metodlar tarafından erişilebilmesini sağlayabiliriz.

Ayrıca, generic olarak tanımlanacak olan **DBSet'i** private olarak belirleyeceğim. **Readonly** anahtar kelimesiyle, bu alana sadece **constructor** veya bu anda değer atayabileceğimizi ifade ediyoruz. Farklı durumlarda set edilemeyeceğini belirtmek için bu önemlidir. **DBSet** tanımlamasını ekstra bir parametre olarka constructor'a yazmama gerek yok, çünkü zaten **context** içinden **_context.Set<T>** yapıldığında bu alan gelecektir. Dönüş değerine bakıldığında ise **DBSet** olduğunu görebilirsiniz. Örnek kod aşağıdaki gibidir.

```
public class GenericRepository<T> : IGenericRepository<T> where T : class
 {
    private readonly DbSet<T> _entities;
    private readonly DbContext _context;

    public GenericRepository(DbContext context)
    {
      _context = context ?? throw new ArgumentNullException(nameof(context));
      _entities = context.Set<T>();
    }

    public async Task<T> GetByIdAsync(int id)
    {
      return await _entities.FindAsync(id);
    }

    public IQueryable<T> GetAll(Expression<Func<T, bool>> expression = null)
    {
      return _dbSet.AsNoTracking().Where(expression).AsQueryable();
    }
```

```
    public IQueryable<T> Where(Expression<Func<T, bool>> expression)
    {
      return _entities.Where(expression);
    }

    public async Task AddAsync(T entity)
    {
      await _entities.AddAsync(entity);
    }

    public async Task AddRangeAsync(IEnumerable<T> entities)
    {
      await _entities.AddRangeAsync(entities);
    }

    public void Update(T entity)
    {
      _entities.Update(entity);
    }

    public void Remove(T entity)
    {
      _entities.Remove(entity);
    }

    public void RemoveRange(IEnumerable<T> entities)
    {
      _entities.RemoveRange(entities);
    }
  }
}
```

GetAll İçin Ekstra Bir Not!

Burada **AsNoTracking**() metodu, çekilen verilerin Entity Framework tarafından bellekte takip edilmemesini ve değişikliklerin izlenmemesini sağlar. Bu özellik, performans açısından avantajlıdır, özellikle büyük veri setleri ile çalışırken.
Çünkü, **AsNoTracking**() kullanıldığında, çekilen veriler sadece okunur ve bellekte bir

önbelleğe alınmazlar. Bu durum, verilerin anlık durumlarını takip etmeye gerek olmadığında performans kazancı sağlar.

```
public IQueryable<T> GetAll(Expression<Func<T, bool>> expression = null)
    {
       return _dbSet.AsNoTracking().Where(expression).AsQueryable();
    }
```

GetById İçin Ekstra Bir Not!

FindAsync metodu, belirtilen id'ye sahip nesneyi bulmak için kullanılır. Bu metot, özellikle birincil anahtar (primary key) üzerinden nesneleri çekmek için tasarlanmıştır. Birden fazla primary key durumunda, bu metot üzerinden kullanmak mümkündür.

```
public async Task<T> GetByIdAsync(int id)
    {
       return await _entities.FindAsync(id);
    }
```

Remove İçin Ekstra Bir Not!

Remove metodu, bir entity'nin durumunu **EntityState.Deleted** olarak işaretler ve bu şekilde belirtilen durumdaki tüm entity'leri, bir sonraki **SaveChanges** çağrısında veritabanından siler. **EntityState.Deleted** durumu, silinecek bir entity'yi işaretlemek için kullanılır ve bu durumda SaveChanges çağrılana kadar veritabanından silme işlemi gerçekleşmez. Yorum satırında belirtilen **_context.Entry(entity).State = EntityState.Deleted**; ifadesi, EntityState'i manuel olarak değiştirme seçeneğini sunar, ancak örnekte mevcut kullanımı zaten yeterli olduğu için şu anki durumda gerekli değildir.

```
public void Remove(T entity)
    {
      // _context.Entry(entity).State = EntityState.Deleted;
       _entities.Remove(entity);
    }
```

UnitOfWork

Core katmanında oluşturulan **IUnitOfWork** interfacenin implementasyonu bu bölümde gerçekleştirilir. İlk olarak, **UnitOfWorks** adında bir klasör oluşturulur ve bu klasörün içine **UnitOfWork** adında bir sınıf eklenir. Bu sınıf, **IUnitOfWork** interfacenin bir implementasyonunu sağlayacak şekilde tasarlanacaktır. Bu arada, sınıfın constructor'ını oluşturarak **AppDbContext'yi** parametre olarak alınır. Interface üzerinde bir **Commit** bir

de **CommitAsync** alanları vardı .Mümkün oldukça **async** kodunu çağıracağız. Ancak herhangi bir kullanım senaryosunda asenkron olmayanı da koda dahil ettik.

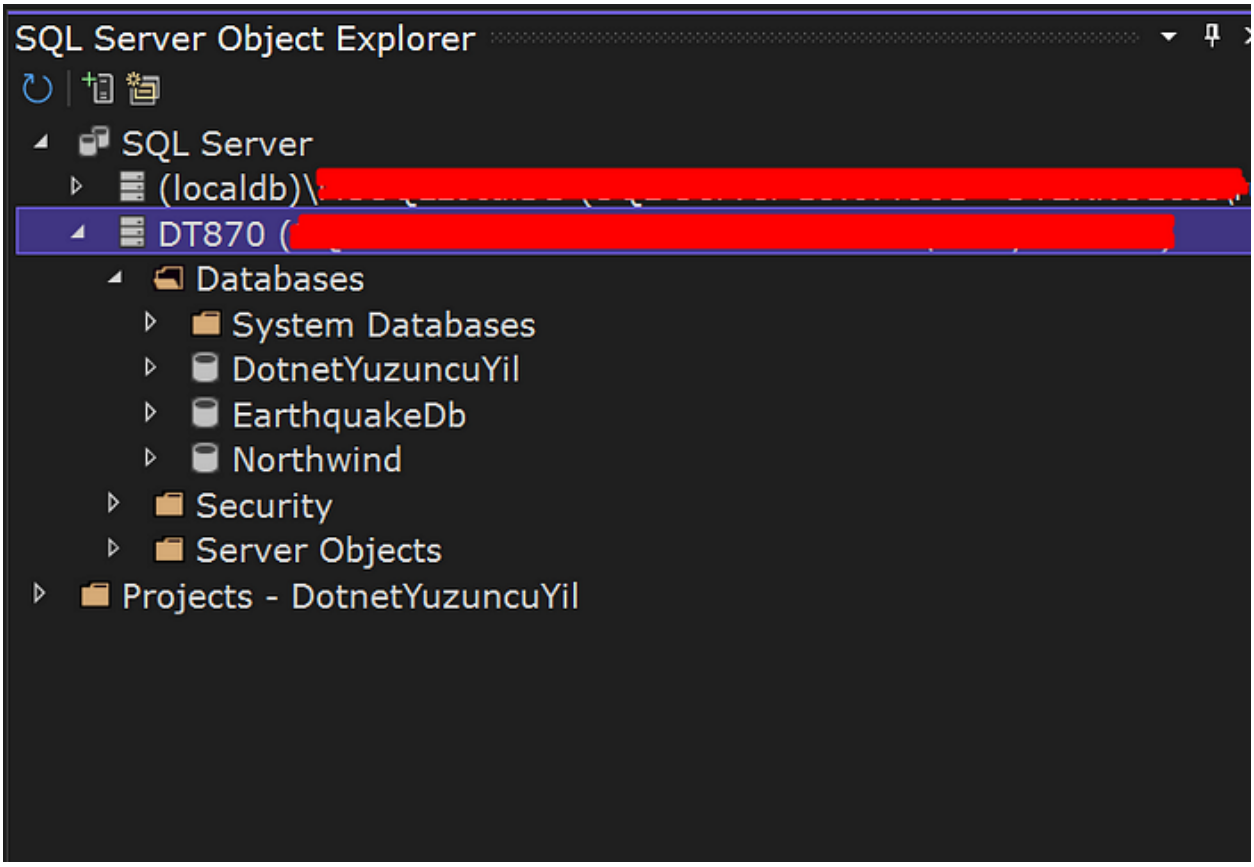
```
using System;
using System.Threading.Tasks;

namespace YourNamespace
{
  public class UnitOfWork : IUnitOfWork
  {
    private readonly AppDbContext _context;

    public UnitOfWork(AppDbContext context)
    {
      _context = context ?? throw new ArgumentNullException(nameof(dbContext));
    }

    public void Commit()
    {
      _context.SaveChanges();
    }

    public async Task CommitAsync()
    {
      await _context.SaveChangesAsync();
    }
  }
}
```

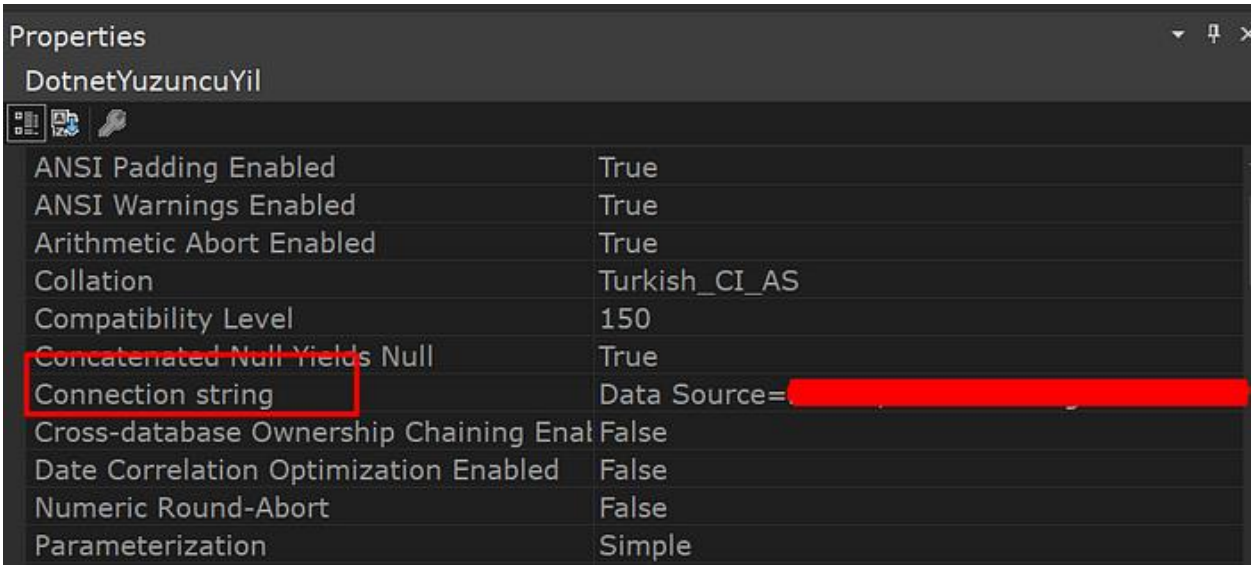
Migrations

Kodlarınızdaki entity'ler ile SQL Server'daki tabloların uyumlu olmasını sağlayan bir araçtır. Uygulama içinde her bir entity'nin güncel durumunu veritabanına yansıtabiliriz. Kodun ilerleyen aşamalarında, yeni özellik eklerseniz, bu özelliklerin de tabloya otomatik olarak güncellenmesini sağlamak için **ORM** araçlarının genellikle bu tür Migration (geçiş) araçları bulunmaktadır.

Veri tabanı yolu nasıl alırım ?

View bölümünde **“SQL Server Object Explorer”** adında bir alanım bulunuyor. Burayı açtığınızda, bir veritabanının yolunu almak için ilgili veritabanına sağ tık yapın ve “**Properties**” seçeneğini seçin. Bu seçeneğe tıkladığınızda, ilgili veritabanının özellikleri penceresi açılacaktır.





ConnectionString’i nereye dahil edeceğim ?

appsettings.json

API katmanında **appsettings.json** içerisine eklemem gerekecek.
Bir **ConnectionStrings** bölümü seçeceğim ve **SqlConnection** olarak adlandıracağım. Bağlantı için kullanıcı adı ve şifre gibi bilgiler de istenebilir. Eğer bu işlemi Windows tarafında yapıyorsanız, alınan bağlantıda bu bilgiler bulunmaz. Ancak, SQL Server Authentication seçeneğini kullanırsanız bu bilgileri eklemeniz gerekir. **"Initial Catalog"** durumu ise tabloları oluşturmak istediğim katalogdur.

```
{
 "ConnectionStrings": {
  "SqlConnection": "Server=your_server_name;Database=your_database_name;User Id=your_username;Password=your_password;"
 },
 "Logging": {
  "LogLevel": {
   "Default": "Information",
   "Microsoft": "Warning",
   "Microsoft.Hosting.Lifetime": "Information"
  }
 },
 "AllowedHosts": "*"
}
```

Program.cs Üzerindeki Eklemeler

SQLConnection ve AdDbContext.cs'in Belirtilmesi

Program.cs dosyasında **AppDbContext** ekledikten sonra, EF Core'un bu connection string'i kullanacağına dair bilgi vermem gerekiyor. Dotnet 6 ile bildiğiniz üzere **Startup.cs** ortadan kalktı ve **Program.cs**'e taşındı. Program.cs üzerinde **AppDbContext'i** ekledikten sonra, bu bağlantının SQL Server'ı kullanacağını belirtmem gerekir. **Builder** üzerinden bu bağlantıya erişip, **appsettings.json'**daki Connection String'i kullanmak için **GetConnectionString** metodu kullanılacaktır. Migration dosyaları Repository katmanı üzerinde oluşturulacaktır ve **AppDbContext,** repository katmanında bulunacaktır. O nedenle **AppDbContext**'in bulunduğu **Assembly**'i API tarafında uygulamaya bildirmem gerekemektedir. Bunun için options ile işlem yapmam gerekmektedir. Options, **MigrationAssembly** metodu kullanılarak assembly adını bu kısımda projenin adı olarak verebilirim(Örn("Repository)). Ancak bu Repository katmanının adı ileride değişirse sorun olabilir. O nedenle "**Assembly**" adlı sınıf

içerisindeki **GetAssembly** metodu ile bir tip verme işlemi gerçekleştirilir ve **typeof** olarak **AppDbContext'in Assembly'si** alınır. Yani **AppDbContext** neredeyse o projenin ismi **GetName()** metodu ile direkt olarak alınır. Böylece repository ismini alarak, tip güvenli bir çalışma gerçekleştirmiş olunur.

```
builder.Services.AddDbContext<AppDbContext>(x=>
{
  x.UseSqlServer(builder.Configuration.GetConnectionString("SqlConnection"), option =>
  {

option.MigrationsAssembly(Assembly.GetAssembly(typeof(AppDbContext)).GetName().Name);
  });
});
```

UnitOfWork Scope Belirtilmesi

```
builder.Services.AddScoped<IUnitOfWork,UnitOfWork()>;
```

GenericRepository Scope Belirtilmesi

Generic olduğu için typeof ile belirtilmesi gerekir.

```
builder.Services.AddScoped<typeof(IGenericRepository<>),typeof(GenericRepository<>)>;
```

Bi küçük Migration Denemesi!

Package Manager Console üzerinden, proje olarak repository projesini seçerek **add-migration initial** komutunu kullanılır. Ancak başlangıç projesi Package Manager Console üzerinden "**Repository**"e denk gelmelidir. Ancak projenin genelinde "**Web**" projesinin seçili olduğuna dikkat edin. Bu web projesi içerisinde de ilgili paketlerin yüklü olması gerekmektedir.
(**Microsoft.EntityFrameworkCore,Microsoft.EntityFrameworkCore.Design,Microsoft.EntityFrameworkCore.Tools**)

```
add-migration initial
```

```
using Microsoft.EntityFrameworkCore.Migrations;

namespace YourNamespace.Migrations
{
  public partial class Initial : Migration
```

```
  {
    protected override void Up(MigrationBuilder migrationBuilder)
    {
      // Buraya eklenen kodlar, veritabanını güncellediğinizde çalışacak olan kısımdır.
      // Tablo ekleme, sütun ekleme, indeks ekleme gibi işlemler burada yapılır.
      // Örnek:
      // migrationBuilder.CreateTable(
      //    name: "SampleTable",
      //    columns: table => new
      //    {
      //       Id = table.Column<int>(nullable: false),
      //       Name = table.Column<string>(nullable: true),
      //    },
      //    constraints: table =>
      //    {
      //       table.PrimaryKey("PK_SampleTable", x => x.Id);
      //    });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
      // Buraya eklenen kodlar, veritabanı geri alındığında çalışacak olan kısımdır.
      // Up metodundaki işlemlerin tam tersini gerçekleştirir.
      // Örnek:
      // migrationBuilder.DropTable(
      //    name: "SampleTable");
    }
  }
}
```

Yukarıdaki örnek bir migrationda, **Initial** adlı bir migration sınıfı oluşturulmuş durumdadır. Bu sınıf, **Migration** sınıfından türemekte ve içerisinde **Up** ve **Down** metodlarını barındırmaktadır.

- **Up metodu:** Veritabanını güncellediğinizde çalışacak olan kısımdır. Bu bölüme, tablo ekleme, sütun ekleme, indeks ekleme gibi veritabanı değişiklikleri eklenir. Örnek kodlarda gösterildiği gibi **CreateTable** metodu kullanılarak yeni bir tablo eklemek mümkündür.

- **Down metodu:** Veritabanı geri alındığında çalışacak olan kısımdır. **Up** metodundaki işlemlerin tam tersini gerçekleştirir. Örnekte gösterildiği gibi **DropTable** metodu kullanılarak bir tablonun silinmesi gibi işlemler gerçekleştirilebilir.

Bu kodlar, veritabanınızı güncellediğinizde ve geri aldığınızda nasıl davranacağını tanımlar. Gerçek projenize uyacak şekilde, tablolarınızı ve ilişkilerinizi düşünerek ilgili migration dosyalarını güncellemelisiniz.

AppDbContextSnapshot ne olur ki acaba ?

AppDbContextSnapshot, Entity Framework Core tarafından otomatik olarak oluşturulan bir sınıftır ve genellikle **Migrations** klasörü altında bulunur. Bu sınıf, veritabanındaki şema (schema) ve yapısal değişiklikleri takip etmek için kullanılır.

Her bir migration oluşturulduğunda, **AppDbContextSnapshot** sınıfı da güncellenir. Bu sınıf, veritabanındaki tabloların ve ilişkilerin anlık bir görüntüsünü sunar. Yani, veritabanındaki her değişiklikte, bu snapshot sınıfı güncellenir ve son durumu temsil eder.

```
// Örnek bir AppDbContextSnapshot sınıfı
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;

[DbContext(typeof(AppDbContext))]
[Migration("20211217000000_InitialMigration")]
partial class AppDbContextSnapshot : ModelSnapshot
{
  protected override void BuildModel(ModelBuilder modelBuilder)
  {
    modelBuilder
      .HasAnnotation("ProductVersion", "6.0.100");

    // Veritabanındaki tablolar ve ilişkiler burada temsil edilir
    modelBuilder.Entity("YourNamespace.Models.SampleTable", b =>
    {
      b.Property<int>("Id")
        .ValueGeneratedOnAdd()
        .HasColumnType("int");

      b.Property<string>("Name")
        .HasColumnType("nvarchar(max)");
```

```
        b.HasKey("Id");

        b.ToTable("SampleTable");
    });
  }
}
```

Bu örnek, **SampleTable** adlı bir modelin (**AppDbContext** sınıfındaki bir DbSet'e karşılık gelen) veritabanındaki temsilini
göstermektedir. **AppDbContextSnapshot** sınıfındaki **BuildModel** metodu, veritabanındaki yapıyı yansılar ve bu sayede bir migration işleminin ardından veritabanı şemasının nasıl göründüğünü gösterir.

Bu snapshot sınıfı, Entity Framework Core'un veritabanı şemasını yönetmesine yardımcı olur ve gelecekteki migration'ları oluştururken, mevcut durumu bilmesini sağlar.

Peki "update-database"ne işe yarar ?

update-database komutu, **Entity Framework Core**'un **Migrations** sistemi üzerinden çalışan ve modelde yapılan değişiklikleri bir veritabanına uygulayan bir komuttur. Bu komut, belirli bir migration'ın **Up** metodunu çalıştırarak modeldeki değişiklikleri veritabanına entegre eder. Yani, yeni tablolar eklemek, sütunları güncellemek, indeksler oluşturmak gibi veritabanı şemasındaki değişiklikleri uygular. Aynı zamanda, **AppDbContextSnapshot** adlı dosyayı güncelleyerek, mevcut veritabanı şemasını temsil eder. **update-database** komutu, geliştiricilere modeldeki değişiklikleri kolayca veritabanına entegre etme ve uygulama ile veritabanı arasındaki uyumsuzlukları giderme imkanı sağlar. Bu sayede, uygulama geliştikçe ve modelde değişiklikler yapıldıkça, veritabanının da güncellenmesi ve bu değişikliklere uygun hale getirilmesi kolaylaşır.

```
update-database
```

Bu işlemleri tamamlamanız ile beraber artık veri tabanında tablolarınız oluşmuş olacaktır.

Bugünlük benim anlatacaklarım bu kadar!

Bir sonraki yazıda görüşmek üzere :)

KAYNAKÇA

- https://www.youtube.com/watch?v=r-RUY2caw3s
- https://www.youtube.com/watch?v=xJC7ItRoEbw

- https://www.youtube.com/watch?v=Srp1iyZu-ww&pp=ygUNbi1sYXllciBwcm9qZQ%3D%3D
- https://www.udemy.com/course/asp-net-core-api-web-cok-katmanli-mimari-api-best-practices/
- https://www.gencayyildiz.com/blog/c-ta-n-tier-architecturecokn-katmanli-mimari/