N-Tier Architecture (Çok Katmanlı Mimari) Yapısını Keşfedelim!

Bir zamanlar Dijital Şehir adında gizemli bir yerde, İnsan Sistemi adlı karmaşık bir yazılım yaşıyordu. Bu yazılım, **Çok Katmanlı** bir yapıya sahipti. Bir gün bu dijital dünyayı keşfetmeye kararlı bir yazılım kaşifi ortaya çıktı.

İlk olarak kaşif **Presentation Layer**'a **(Sunum Katmanı)** rastladı. Bu katman renkli bir meydan gibiydi ve kullanıcı arayüzünü temsil ediyordu. İnsanlar buraya girerek duygusal verilerini ifade ediyor ve dış dünyayla iletişim kuruyordu. Kaşif bu katmandan ayrılıp diğer katmanları keşfetmeye devam etti.

Karşısına çıkan ikinci katman **İş Katmanıydı (Business Layer)**. Bu katman projenin kalbi olarak düşünülüyordu. Kaşif burada temel iş süreçlerinin ve veri işleme mantığının bulunduğunu fark etti. Karmaşık algoritmalar ve iş kurallarıyla dolu bir labirent gibiydi. Kaşif bu katmanı inceleyerek iç dünyasının işleyişini anlamaya çalıştı.

Üçüncü katman **Veri Depolama Katmanıydı (Data Access Layer)**. Bu katman projenin belleğini temsil ediyordu. Burada veriler depolanıyor, yönetiliyor ve çeşitli katmanlara dağıtılıyordu. Kaşif kendi duygusal verilerini ve düşünce algoritmalarını içeren verileri inceleyerek bu katmanı ziyaret etti.

Her katmanı keşfettikçe kaşif, iç dünyasının ihtiyaçlarına uygun olarak katmanlı mimariyi anladı. Katman sayısını artırıp azaltarak içsel dünyasını daha etkili bir şekilde yönetmeye odaklandı.

Kaşif, Dijital Şehir'den döndüğünde yazılımın yüzeyinde dolaşan diğer kodlar ona içsel dünyasını sormadılar ancak o artık kendi iç kod mimarisini anlamış, iyileştirmiş ve dengelemiş bir şekilde yaşamaya devam etti. İnsanlar dijital dünyanın yüzeyinde dolaşırken, Kaşif'in içsel keşiflerinden habersiz olarak devam ettilerancak o artık kendi iç kod şehrini yönetebilen bir yazılım olarak bilinir hale gelmişti.

Bugün N-Katmanlı bir projenin derinliklerine inerek, her bir katmanın içerisinde yer alan alanları detaylı bir şekilde inceleyeceğiz. Yazıyı okurken hem keyif almanızı umuyorum hem de projelerinizdeki katman yapılarını daha iyi anlamanıza katkı sağlamasını diliyorum. Keyifli okumalar :)

Neden Çok Katmanlı Mimari Tercih Edilmeli?

Çok katmanlı mimari, bir yazılım sisteminin farklı işlevsel katmanlara ayrılarak tasarlandığı bir yaklaşımdır. Bu tür bir mimari, genellikle büyük ve karmaşık sistemlerin geliştirilmesinde tercih edilir.

Avantajları

• Kod tekrarını önler ve proje bakımı kolaylaşır.

• Farklı katmanlar farklı ekipler tarafından geliştirilebilir.

• Farklı katmanlar farklı teknolojiler kullanılabilir.

•Bir katmanın performansını artırmak veya ölçeklendirmek, diğer katmanları etkilemeden gerçekleştirilebilir.

• Her katman diğer katmanlardan bağımsız olarak yeniden kullanılabilir.

Dezavantajları

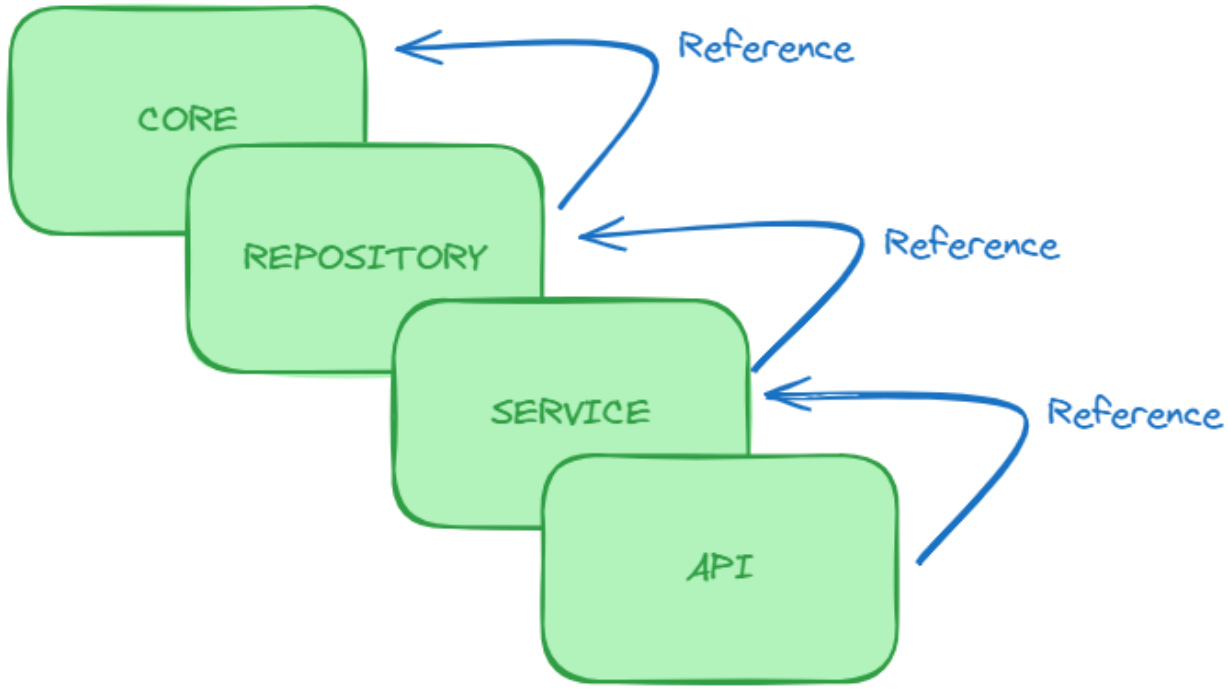
• Çok katmanlı mimari, sistem tasarımını karmaşık hale getirebilir. Bu karmaşıklık, bazen yönetim ve anlama zorluklarına yol açabilir.

• Her katmanın tasarımı için doğru kararları vermek, deneyim ve uzmanlık gerektirir. Yanlış tasarım kararları, sistemdeki diğer katmanları etkileyebilir.

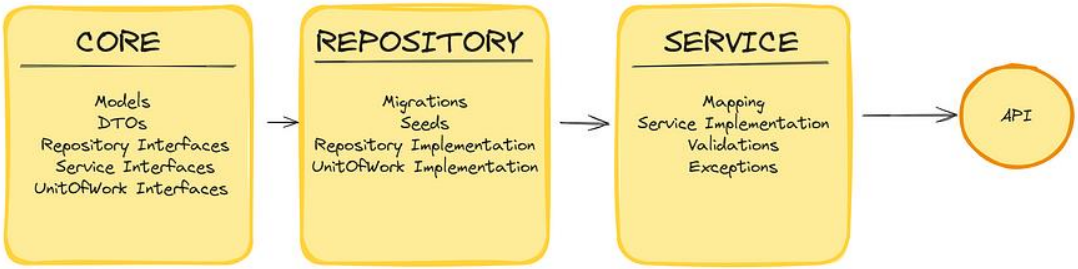
• Çok katmanlı mimariler genellikle daha uzun uygulama başlatma sürelerine neden olabilir. Bu durum, hızlı başlatma gerektiren uygulamalarda sorunlara yol açabilir.

Temel Katmanlar

Çok katmanlı mimari genellikle **sunum katmanı** (presentation layer), **iş mantığı katmanı** (business logic layer) ve **veri erişim katmanı** (data access layer) olarak **üç temel katmandan** oluşur. Ben alternatif olarak Core Repositoryve Serviceisimlendirmelerini kullanıyor olacağım.



• Repsository katmanı Core katmanını referans alır.
• Service katmanı Repository katmanını referans alır.
• API katmanı da Service katmanını referans alır.



Her katmanın kendine göre görevleri vardır. Bu katmanları detaylı bir şekilde inceleyelim.

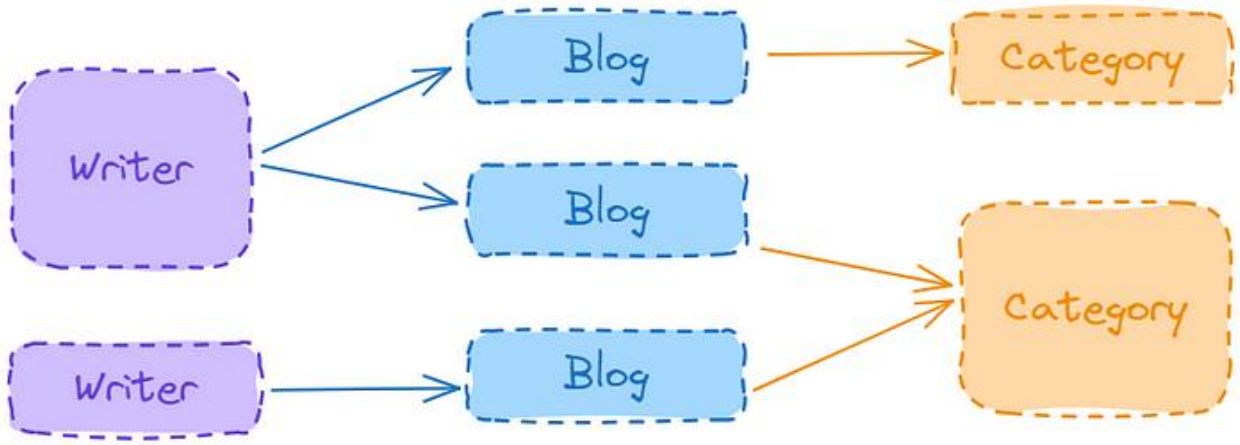
CORE KATMANI

Core katmanı, projenin temelini oluşturan sınıfları içerir ve özel iş kurallarını barındırmaz. Bu katman, yazılımın sürdürülebilirliğini artırarak genel proje yapısını düzenler ve proje boyunca tekrar kullanılabilir bir yapı sunar.

Projenin genelinde kullanılan:
Modeller, DTO nesneleri, servis ve repository interfaceleri, aynı zamanda **unit of work design pattern'ı için interfaceler** yer almaktadır.

NOT: Bir sınıfın veritabanında karşılığı varsa bu sınıf "**entity**" olarak adlandırılır ancak her sınıf entity olmayabilir.

Entityleri Oluşturma



Bir blog yalnızca bir kategoriye aittir ama bir kategori içinde birden çok blog olabilir ve bir blog bir yazar tarafından yazılmıştır.Bir yazarın birden çok blog'u olabilir.

N Katmanlı mimariyi bu yapı üzerinde inşa edelim.

Base Entity

Base Entity, yazılım projelerinde diğer entity'lerin *ortak özelliklerini* içeren bir ana sınıf veya arayüzdür. Bu sayede kod düzeni, veritabanı tasarımı ve projedeki tutarlılık daha etkili ve esnek bir şekilde sağlanabilir.

Kendi başına bir örneğinin oluşturulmaması için abstractbir base entity oluşturulur ve *diğer entityler bu BaseEntityden türer*

```
public abstract class BaseEntity
{
    public int Id { get; set; }
    public DateTime CreatedDate { get; set; }
    public DateTime? UpdatedDate { get; set; }
}
```

CreatedDate özelliği, nesnenin oluşturulma tarihini temsil eder ve bu değer her zaman atanır ancak UpdatedDate özelliği, nesnenin güncellenme tarihini temsil eder, nesne henüz güncellenmemişse null olabilir. Bu durumu ifade etmek için DateTime? kullanılır.

```
public class Blog : BaseEntity
{
    public string BlogTitle { get; set; }
    public string BlogContent { get; set; }
```

```
    public int CategoryID { get; set; }     //foreign key
    public Category Category { get; set; }   //navigation property

    public int WriterID { get; set; }    //foreign key
    public Writer Writer { get; set; }  //navigation property
}
```

CategoryID ve WriterID: Category ve Writer sınıfları ile ilişki kurmak için kullanılan foreign key alanlarıdır.

```
  public class Category : BaseEntity
  {
      public string CategoryName { get; set; }

      public ICollection<Blog> Blogs { get; set; }
  }
```

ICollection<Blog>kategoriyle ilişkilendirilmiş blogları tutmak için kullanılan bir koleksiyon yapısıdır. Blogs o kategoriye ait bir veya birden fazla blogu içeren bir koleksiyondur.

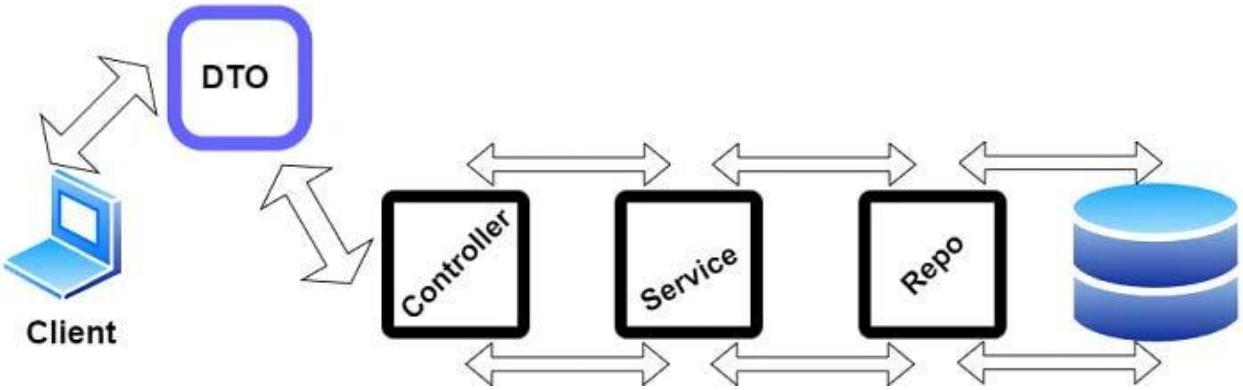
```
  public class Writer : BaseEntity
  {
      public string WriterName { get; set; }
      public string WriterMail { get; set; }
      public string WriterPassword { get; set; }

      public ICollection<Blog> Blogs { get; set; }
  }
```

ICollection<Blog> Blogs Writer sınıfının bir koleksiyonu olup, bir yazarın yazdığı blogları temsil eder.

DTO (Data Transfer Object) Nedir?

DTO'ların kullanılma nedeni, bir istemcinin bir sunucudan veri alması veya sunucuya veri göndermesi gerektiğinde, veri transfer işlemini düzenli ve etkin bir şekilde gerçekleştirmeleridir.

DTO nesneleri, API tabanlı geliştirmelerde vazgeçilmez unsurlardır. Çünkü, Entity sınıflarını doğrudan apiler aracılığıyla dış dünyaya açmak, güvenlik açıklarına neden olabilir.




```
public abstract class BaseDto
{
    public int Id { get; set; }
    public DateTime CreatedDate { get; set; }
}

public class WriterDto:BaseDto
{
    public string WriterName { get; set; }
    public string WriterMail { get; set; }
}
```

İlgili verileri taşıyan bir DTO'da **iş mantığı olmaz**, sadece veri transferi amaçlanır.

"Writer*Dto*" sınıfı, "*Writer*" modeline benzer özellikleri içerir ancak iş mantığı veya veritabanı operasyonları içermez. Genellikle bu tür bir DTO, bir API üzerinden bir istemciye veri göndermek veya istemciden almak için kullanılır. Bu sayede sadece ***gerekli verilerin*** transfer edilmesi sağlanır ve gereksiz bilgilerin açığa çıkması önlenir.

Repository Interfaces

Repository tasarım deseni, genellikle veritabanı işlemlerini soyutlayarak uygulamaların veri tabanı ile etkileşimini düzenleyen bir tasarım desenidir.

Repository'ler, veritabanına erişimi soyutlar ve uygulamanın geri kalan kısmıyla arasında bir arayüz görevi görerek veri tabanı işlemlerini yönetir. Bu, uygulama katmanlarının doğrudan veri tabanına erişmek yerine bu soyutlama aracılığıyla işlemlerini gerçekleştirmelerini sağlar.

Bu sayede uygulama katmanları, veri tabanıyla olan bağımlılıklarını azaltabilir.

IGenericRepository Interface'i

Generic yapı, her bir entity için temel CRUD (Create, Read, Update, Delete) işlemlerini tekrar kullanılabilir bir şekilde uygulamamızı sağlar. Daha sonra bu interface'i implemente eden bir sınıf belirli bir varlık türü için veri tabanı işlemlerini sağlar.

```
public interface IGenericRepository<T> where T : class
{
    Task<T> GetByIdAsync(int id);
    IQueryable<T> GetAll();
    IQueryable<T> Where(Expression<Func<T, bool>> expression);
    Task AddAsync(T entity);
    Task AddRangeAsync(IEnumerable<T> entities);
    void Update(T entity);
    void Remove(T entity);
    void RemoveRange(IEnumerable<T> entities);
}
```

IQueryable Neden Kullanılır ?
Bu interface'in kullanımının temel amacı, sorguların ***veri tabanına gitmeden önce*** bir sorgu ağacı oluşturmasını sağlamaktır. Bu da uygulama performansını artırır ve veri tabanı etkileşimini daha etkili hale getirir.

Bir veri tabanı sorgusu genellikle bir dizi filtreleme, sıralama veya gruplama işleminden oluşur.

Bu tür sorgular, ToList() veya ToListAsync() çağrıldığında veri tabanına iletilir.

Asenkron Metotlar
Asenkron metotlar, genellikle **uzun süren işlemleri** gerçekleştirmek için kullanılır.
Bu metotlar, bir işlemin tamamlanmasını beklerken diğer işlemlerin devam etmesini sağlar.
Async metotlar genellikle Asyncekini alır ve genellikle Task veya Task<T> türünde bir nesne döndürür.

Update ve Remove Metotları:
Update veya Remove gibi metotlar genellikle veri tabanı üzerinde **kısa süren** işlemleri içerir. Bu tür işlemler genellikle anlık olarak gerçekleşir ve uzun süreli beklemeyi gerektirmez. Dolayısıyla bu tür metotlar genellikle asenkron olarak işaretlenmez çünkü asenkron kullanımı daha çok uzun süren işlemleri yönetmek için kullanılır.

Add Metodu:
Add gibi metotlar genellikle yeni bir öğe eklemek gibi işlemleri içerir. Bu işlemler özellikle büyük veri setleri üzerinde çalışılıyorsa zaman alabilir. Bu nedenle Add gibi işlemler asenkron olarak işaretlenebilir böylece uzun süren işlemler arka planda gerçekleştirilir.

Service Interfaces

Servisler genellikle iş mantığı veya uygulama seviyesindeki belirli görevleri yerine getiren ve bu görevleri **merkezi** bir şekilde yöneten bileşenlerdir. Service interfaceleri, bu servislerin kullanılabilir arayüzlerini tanımlar ve bu arayüzler sayesinde servislerle etkileşim kurulmasını sağlar. Bu arayüzler servislerin sağladığı işlevselliği soyutlar, böylece uygulamanın farklı bileşenleri bu servislerle bağımsız ve etkili bir şekilde çalışabilir.

IService Interface'i

```
public interface IService<T> where T : class
{
   Task<T> GetById(int id);
   Task<IEnumerable<T>> GetAllAsync();
   IQueryable<T> Where(Expression<Func<T, bool>> expression);
   Task<T> AddAsync(T entity);
   Task<IEnumerable<T>> AddRangeAsync(IEnumerable<T> entities);
   Task UpdateAsync(T entity);
   Task RemoveAsync(T entity);
   Task RemoveRangeAsync(IEnumerable<T> entities);
}
```

Service katmanındaki Update ve Remove işlemlerinin asenkron ve Task döndüren metotlar olarak işaretlenmesinin temel nedeni, bu işlemlerin genellikle veri tabanına yansıtma operasyonları olmasıdır. Bu tür veritabanı işlemleri zaman alabilir ve asenkron bir yapı kullanılarak bu süreç daha verimli bir şekilde yönetilebilir.

UnitOfWork Design Pattern

Unit of Work, bir tasarım deseni olup genellikle veri tabanı işlemlerini gruplamak, yönetmek ve koordine etmek amacıyla kullanılır.
Bu desen, bir iş birimi içindeki bir dizi işlemi tek bir iş birimi olarak ele alır. Unit of Work deseni, aynı zamanda bir dizi işlemi birleştirerek bu işlemleri tek bir "**transaction**" (işlem) içinde işleyebilir ve tutarlı bir durumu korumaya yardımcı olabilir.

```
public interface IUnitOfWork
{
   Task CommitAsync();
   void Commit();
}
```

Repository Katmanı

Repository katmanı, veri tabanı veya diğer veri kaynaklarına erişimi yönetir. Veri modeli ile ilgili temel CRUD (Create, Read, Update, Delete) işlemlerini gerçekleştirir.

Veri tabanı sorgularını içerir ve bu sorgular aracılığıyla veri tabanından veri çekme veya veri güncelleme gibi işlemleri yönetir.

Repository katmanı, Core katmanını referans alır. Bu, Core katmanındaki modelleri ve diğer yapıları kullanarak veri işlemlerini gerçekleştirir.

AppDbContext

AppDbContextveri tabanı işlemlerini yöneten bir sınıftır. Bu sınıf, Entity Framework Core veya Entity Framework gibi *ORM* (Object-Relational Mapping) araçlarıyla etkileşimde bulunarak veri tabanı ile iletişim kurar.

Microsoft.EntityFrameworkCore,
Microsoft.EntityFrameworkCore.Design,
Microsoft.EntityFrameworkCore.Tools,
Microsoft.EntityFrameworkCore.SQLServer

kütüphanelerinin yüklenmesi gerekir.

AppDbContext sınıf DbContext sınıfından **türetilmeli** ve **constructor** oluşturulmalıdır. Her bir entity için DbSet içermelidir.

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

   public DbSet<Blog> Blogs { get; set; }
   public DbSet<Category> Categories { get; set; }
   public DbSet<Writer> Writers { get; set; }


   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
      base.OnModelCreating(modelBuilder);
   }
}
```

ApplyConfigurationFromAssembly metodu, Fluent API konfigürasyonlarını temsil eden ve IEntityTypeConfiguration<TEntity> arayüzünü uygulayan ayrı sınıfların kullanılmasını sağlar. Bu metod, mevcut assembly içindeki tüm Fluent API konfigürasyon sınıflarını otomatik olarak tespit eder ve uygular böylece her varlık türü için özel konfigürasyonları tek tek belirtmek yerine bu sınıfları tek bir yerden toplu bir şekilde tanımlamanıza olanak sağlar.

Konfigürasyonlar

Fluent API, veri tabanı modelini belirli bir biçimde özelleştirmek için kullanılan bir dizi metot ve ayarı içerir. Bu metotlar genellikle OnModelCreating metodunda kullanılır ve veri tabanındaki tabloların, ilişkilerin, anahtarların ve diğer özelliklerin oluşturulma biçimini belirler.

Bu konfigürasyonları daha düzenli bir şekilde organize etmek için genellikle “Configurations” adında bir klasör oluşturulur. Her varlık türü için bir konfigürasyon sınıfı eklenir. Örneğin, “BlogConfiguration” adında bir sınıf oluşturabilir ve bu sınıfın IEntityTypeConfiguration<Blog> arayüzünü uygulamasını sağlayarak "Configure" metodunu içerisinde Blog ile ilgili tüm konfigürasyonları ekleyebiliriz.

```
public class BlogConfiguration : IEntityTypeConfiguration<Blog>
{
  public void Configure(EntityTypeBuilder<Blog> builder)
  {

    builder.HasKey(x => x.Id); // Primary Key (PK)

    builder.Property(t => t.Id)  //Pk 1-1 arttırma işlemi
     .UseIdentityColumn();

    builder.Property(x => x.BlogTitle)
    .HasMaxLength(100)
    .IsRequired();

    builder.Property(x => x.BlogContent)
    .IsRequired();
  }
}
```

Seeds:

- Seed data, bir uygulamanın başlangıç durumunu tanımlayan ve genellikle veri tabanına ön tanımlı verilerin eklenmesini sağlayan veri setlerini ifade eder. Bu veriler, uygulama ilk kez çalıştırıldığında veya belirli bir durumda (örneğin, boş bir veritabanası oluşturulduğunda) kullanılabilir.
- Seed datalarda Id'leri manuel olarak kendimiz atamalıyız.

```
public class CategorySeed : IEntityTypeConfiguration<Category>
{
   public void Configure(EntityTypeBuilder<Category> builder)
   {
      builder.HasData(
         new Category
         {
            Id = 1,
            CategoryName = "Teknoloji",
            CreatedDate = DateTime.Now,
         },
         new Category
         {
            Id= 2,
            CategoryName = "Yazılım",
            CreatedDate = DateTime.Now,
         },
         new Category
         {
            Id = 3,
            CategoryName = "Trendler",
            CreatedDate = DateTime.Now,
         }

         );
   }
}
```

Repository Implementasyonu:

- Core katmanındaki Repository interfacelerinin implementasyon işlemi burada gerçekleşir. Repository sınıfları Core katmanındaki iş mantığı ile veri kaynakları arasındaki bağlantıyı kurar.

Generic Repository

Core katmanındaki **IGenericRepository** interface'i bu sınıfta implement ederiz. Bu sınıf, tüm varlık türleri için genel CRUD operasyonlarını içerir.

AppDbContext property'si *protected* olarak tanımlandığı için, sadece bu sınıftan türetilmiş sınıflar bu özelliğe erişebilir.

DBSet, GenericRepository sınıfında generic olarak tanımlanmış ve private olarak belirlenmiştir. Readonly anahtar kelimesi bu alanın sadece constructor içinde veya başlatma aşamasında değer alabileceğini belirtir. Yani, bir kez başlatıldıktan sonra değeri değiştirilemez.

```
public class GenericRepository<T> : IGenericRepository<T> where T : class
{
  protected readonly AppDbContext _context;
  private readonly DbSet<T> _dbSet;

  public GenericRepository(AppDbContext context)
  {
    _context = context;
    _dbSet = _context.Set<T>();
  }

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

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

  public IQueryable<T> GetAll()
  {
    return _dbSet.AsNoTracking().AsQueryable();
  }

  public async Task<T> GetByIdAsync(int id)
```

```
    {
        return await _dbSet.FindAsync(id);
    }

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

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

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

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

UnitOfWork Implementasyonu:

- Core katmanındaki UnitOfWork Interface'sinin implementasyon işlemi burada gerçekleşir. UnitOfWork, işlemleri gruplamak, transaction yönetimi sağlamak ve veri kaynakları ile etkileşimi düzenlemek için kullanılır.

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
    }
```

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

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

Migrations:

- Migrations, veri tabanı ile senkronizasyonu sağlar. Veritabanındaki model değişikliklerini izler ve bu değişiklikleri veritabanına uygular.

Migrate işlemi için ilk adım, API katmanındaki appsettings.json dosyasına ConnectionStrings eklemektir.

```
{
  "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": "*"
}
```

Hassas verilerin (örneğin, geliştirme veritabanı bağlantı dizesi, API anahtarları veya diğer gizli bilgiler) güvenli bir şekilde saklanması için API katmanında sağ tıklayarak "**Manage User Secrets**" seçeneğini kullanmanızı öneririm. Bu araç, geliştirme ortamında kullanıcıya özgü gizli verileri güvenli bir şekilde yönetmek için tasarlanmıştır.

Program.cs dosyasına AppDbContext'in eklenmesi gerekmektedir.

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

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

Package Manager Console üzerinden, repository katmanı seçilerek add-migration initial komutu ile ilk migrate işlemi gerçekleştirilir. Daha sonra update-database komutu ile bu migrate işlemi veritabanına uygulanır.

Service Katmanı

Service katmanı, uygulamanın iş mantığı kurallarını uygular. Bu katman, Repository katmanından alınan verileri kullanarak iş mantığı işlemlerini gerçekleştirir. Aynı zamanda dış dünya ile iletişimi sağlar ve genellikle kullanıcı arayüzü veya diğer sistemlerle etkileşimde bulunur.

Service katmanı, genellikle kullanıcı arayüzü veya diğer sistemlerle etkileşimde bulunur. Kullanıcı taleplerini karşılamak ve diğer sistemlere bilgi aktarmak için kullanılır.

Service katmanı, Repository katmanını referans alır. Bu, iş mantığı kurallarını uygularken veri kaynaklarından alınan verileri kullanmak için gereklidir.

Mapping (AutoMapper)

- Entity sınıfları ile DTO sınıfları arasında *veri dönüşümünü* sağlamak için AutoMapper gibi araçları içerir. Bu, veri tabanından alınan veriyi uygun formata dönüştürmeyi ve kullanıcı arayüzüne iletmeyi sağlar.
- **AutoMapper.Extension.Microsoft.DependencyInjection** paketini eklememiz gerekir..
- Service Katmanına Mapping adında bir kalsör ekleyip, MapProfile adında bir class oluşturuyoruz.

```
public class MapProfile:Profile
{
  public MapProfile()
  {
```

```
        // entity'den dto çevirme
        CreateMap<Blog,BlogDto>().ReverseMap();
        CreateMap<Category,CategoryDto>().ReverseMap();
        CreateMap<Writer,WriterDto>().ReverseMap();

        //dto'dan entity'e çevirme
        CreateMap<BlogDto, Blog>();
        CreateMap<CategoryDto, Category>();
        CreateMap<WriterDto, Writer>();
    }
```

- Program.cs de **AutoMapper** kütüphanesinin konfigürasyon işlemini gerçekletirmek gerekiyor.

```
builder.Services.AddAutoMapper(typeof(MapProfile));
```

Service Implementasyonu

- Core katmanındaki Service Interfaceleri'nin implementasyonlarını içerir. Service sınıfları, iş mantığı kurallarını uygular ve Controller veya diğer katmanlarla etkileşime geçer.

```
 public class Service<T> : IService<T> where T : class
{
    private readonly IGenericRepository<T> _repository;
    private readonly IUnitOfWork _unitOfWork;

    public Service(IGenericRepository<T> repository, IUnitOfWork unitOfWork)
    {
        _repository = repository;
        _unitOfWork = unitOfWork;
    }
    public async Task<T> AddAsync(T entity)
    {
        await _repository.AddAsync(entity);
        await _unitOfWork.CommitAsync();
        return entity;
    }

    public async Task<IEnumerable<T>> AddRangeAsync(IEnumerable<T> entities)
```

```
{
    await _repository.AddRangeAsync(entities);
    await _unitOfWork.CommitAsync();
    return entities;
}

public async Task<IEnumerable<T>> GetAllAsync()
{
    return await _repository.GetAll().ToListAsync();
}

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

public async Task RemoveAsync(T entity)
{
    _repository.Remove(entity);
    await _unitOfWork.CommitAsync();
}

public async Task RemoveRangeAsync(IEnumerable<T> entities)
{
    _repository.RemoveRange(entities);
    await _unitOfWork.CommitAsync();
}

public async Task UpdateAsync(T entity)
{
    _repository.Update(entity);
    await _unitOfWork.CommitAsync();
}

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

```
    }
}
```

BlogService adında bir sınıf oluşturulur ve bu sınıf **Service<Blog>** sınıfından miras alnır ve **IBlogService** interface'si uygulanır.

```
public class BlogService : Service<Blog>, IBlogService
{
    public BlogService(IGenericRepository<Blog> repository, IUnitOfWork unitOfWork) :
base(repository, unitOfWork)
    {
    }
}
```

Sınıfın constructor'ı, iki bağımlılığı enjekte eder: IGenericRepository<Blog> ve IUnitOfWork. Bu bağımlılıklar, genel veritabanı işlemlerini (CRUD operasyonları) ve iş birimi desenini yönetmek için kullanılır. Diğer sınıflar içinde aynısı uygulanır.

Validations:

- Giriş doğrulamalarını içerir. Kullanıcıdan gelen verilerin doğruluğunu kontrol etmek ve gerekirse hata mesajları dönmek için kullanılır.
- **Fluent Validation** kütüphanesini kullanrak gerçekleştirilebilir.
- Service Katmanına Validations adında bir klasör oluşturup tüm entityler için validation'ları burada gerçekleştirebiliriz.

```
public class BlogDtoValidator : AbstractValidator<BlogDto>
{
    public BlogDtoValidator()
    {
        RuleFor(x => x.BlogTitle).NotEmpty().WithMessage("Blog başlığı boş olamaz.")
            .NotNull().WithMessage("Blog başlığı null olamaz.");

        RuleFor(x => x.BlogContent).NotEmpty().WithMessage("Blog içeriği boş olamaz.")
            .NotNull().WithMessage("Blog içeriği null olamaz.");
    }
}
```

Program.cs FluentValidation'un eklenmesi gerekmektedir.

```
builder.Services.AddControllers().AddFluentValidation(x =>
    {x.RegisterValidatorsFromAssemblyContaining<BlogDtoValidator>();});
```

Exceptions Handling:

- Hata yönetimi ile ilgili kodları içerir. Bu, iş mantığı işlemlerinde veya diğer hatalarla ilgili istisnaları ele almayı içerir. Hataların uygun şekilde yönetilmesi, uygulamanın daha sağlam ve güvenilir olmasına katkı sağlar.

API (Presentation) Katmanı

API katmanı, uygulamanın dış dünyayla iletişim kurduğu yerdir ve genellikle HTTP protokolü üzerinden gelen taleplere yanıt verir. Bu katman, gelen HTTP isteklerini alır, işler ve uygun yanıtları oluşturarak istemcilere geri gönderir.

İlk olarak GlobalResultDto adında bir dto oluşturulur.

GlobalResultDto, API tarafından dönen yanıtları standartlaştırmak için kullanılan bir yapıdır. Bu yapı, HTTP durum kodunu, işlemin sonucunu temsil eden veriyi ve açıklayıcı bir mesajı içerir. Kullanılma amacı, istemcilere tutarlı bir API deneyimi sağlamaktır. Örneğin, başarılı bir yanıt veya bir hata durumundaki yanıt, bu yapı üzerinden standart bir formatta iletilir.

```
public class GlobalResultDto<T>
 {
    public T Data { get; set; }

    [JsonIgnore]
    public int StatusCode { get; set; }

    public List<string> Errors { get; set; }

    public static GlobalResultDto<T> Success(int statusCode, T data)
    {
       return new GlobalResultDto<T> { Data = data, StatusCode = statusCode };
    }

    public static GlobalResultDto<T> Success(int statusCode)  //Data göndermek
istemezsek
    {
       return new GlobalResultDto<T> { StatusCode = statusCode };
    }
```

```
    public static GlobalResultDto<T> Fail(int statusCode, List<string> errors)
    {
        return new GlobalResultDto<T> { StatusCode = statusCode, Errors = errors };
    }

    public static GlobalResultDto<T> Fail(int statusCode, string error)
    {
        return new GlobalResultDto<T> { StatusCode = statusCode, Errors = new List<string> { error } };
    }
 }
```

Örneğin hata durumunda:

```
{
  "StatusCode": 404,
  "Data": null,
  "Message": "Belirtilen kaynak bulunamadı."
}
```

```
{
  "StatusCode": 200,
  "Data": { /* Gerçek veri */ },
  "Message": "İşlem başarıyla tamamlandı."
}
```

Sonra NoContentDto sınıfın oluşturulur.

```
 public class NoContentDto
 {
    [JsonIgnore]
    public int StatusCode { get; set; }
    public List<string> Errors { get; set; }
 }
```

Bu yapı özellikle kaynak güncellemeleri veya silmeler gibi durumlarda başarılı bir işlemin gerçekleştiğini ancak özel bir yanıt içeriği gönderilmediğini belirtmek için kullanılabilir.

- *StatusCode:* HTTP durum kodunu belirtir. Ancak JsonIgnore ile işaretlendiği için bu özellik JSON serileştirmesi sırasında dışa aktarılmaz. Yani istemciye gönderilen JSON yanıtında yer almaz.

- *Errors:* Hata durumları için kullanılabilir. Eğer bir hata oluştuysa ve bu hata ile ilgili bilgiler istemciye iletilmek isteniyorsa, bu listede hata mesajları saklanabilir. Bu örnek içinde NoContentDto tipi genellikle hata içermeyecek bir durumu temsil ettiği için bu alan genellikle boş olacaktır.

Daha sonra, API kontrolcüsü olan CustomBasesController sınıfını oluşturuyoruz. Bu sınıf API endpoint'lerinin temelini oluşturur ve HTTP isteklerine yanıt verir.

```
[Route("api/[controller]")]
[ApiController]
public class CustomBasesController : ControllerBase
{
  [NonAction]
  public IActionResult CreateActionResult<T>(GlobalResultDto<T> response)
  {
    if (response.StatusCode == 204)
    {
      return new ObjectResult(null)
      {
        StatusCode = response.StatusCode
      };
    }

    return new ObjectResult(response)
    {
      StatusCode = response.StatusCode
    };

  }
```

- CreateActionResult metodu, API'nin döneceği yanıtları oluşturan genel bir metottur.
- [NonAction] özelliği, Swagger veya benzeri API belgelendirme araçları tarafından görüntülenmeyen ve API'nin dış dünyayla etkileşime girmeyen metodları belirtmek için kullanılır.
- GlobalResultDto<T> tipi, API yanıtlarının genel bir yapısal şablonunu temsil eder.
- Bu metod, GlobalResultDto'ya uygun şekilde yanıt oluşturur ve HTTP durum kodunu belirler.

- BlogsController, CustomBasesController sınıfından miras alır.
  Bu, CreateActionResult metodunu kullanma yeteneği sağlar.

```
public class BlogsController : CustomBasesController
{
  private readonly IMapper _mapper;
  private readonly IBlogService _blogService;

  public BlogsController(IMapper mapper, IBlogService blogService)
  {
    _mapper = mapper;
    _blogService = blogService;
  }

  [HttpGet]
  public async Task<IActionResult> All()
  {
    var blogs = await _blogService.GetAllAsync();
    var blogsDto = _mapper.Map<List<BlogDto>>(blogs.ToList());
    return CreateActionResult(GlobalResultDto<List<BlogDto>>.Success(200, blogsDto));
  }

  [HttpGet("{id}")]
  public async Task<IActionResult> GetById(int id)
  {
    var blog = await _blogService.GetById(id);
    var blogDto = _mapper.Map<BlogDto>(blog);
    return CreateActionResult(GlobalResultDto<BlogDto>.Success(200, blogDto));
  }

  [HttpPost]
  public async Task<IActionResult> Save(BlogDto blogDto)
  {
    var blog = await _blogService.AddAsync(_mapper.Map<Blog>(blogDto));
    var teamDtos = _mapper.Map<BlogDto>(blog);
    return CreateActionResult(GlobalResultDto<BlogDto>.Success(201, blogDto));
  }
```

```
    [HttpPut]
    public async Task<IActionResult> Update(BlogDto blogDto)
    {
        await _blogService.UpdateAsync(_mapper.Map<Blog>(blogDto));
        return CreateActionResult(GlobalResultDto<NoContentDto>.Success(204));
    }
    [HttpDelete("{id}")]
    public async Task<IActionResult> Remove(int id)
    {
        var blog = await _blogService.GetById(id);
        await _blogService.RemoveAsync(blog);
        return CreateActionResult(GlobalResultDto<NoContentDto>.Success(204));
    }
}
```

- All, GetById, Save, Update, ve Remove metodları, belirli HTTP metotlarına (GET, POST, PUT, DELETE) karşılık gelir.
- Bu metotlar, _blogService üzerinden iş mantığı işlemlerini çağırarak, veri tabanı işlemlerini gerçekleştirirler.
- Dönen sonuçlar, GlobalResultDto yapısı içinde düzenlenir ve CreateActionResult metodu kullanılarak uygun HTTP yanıtları oluşturulur.
- AutoMapper kütüphanesi, varolan nesneleri başka bir nesneye dönüştürmek için kullanılır. Bu örnekte, _mapper.Map nesnesi kullanılarak Blog ve BlogDto türleri arasında dönüşümler gerçekleştirilir.

Bu yapı, kodun tekrar kullanılabilir ve sürdürülebilir olmasını sağlar. HTTP isteklerine uygun yanıtların oluşturulması için CreateActionResult metodunun kullanılması kodun temiz ve tutarlı olmasını sağlar.

Projeyi detaylı bir şekilde incelemek isterseniz buradan ulaşabilirsirsiniz..

Kaynakça

- N-Layer Proje Yapısı:Katmanlar Arasında Bir Şehir Hikayesi
- Core Katmanı
- Repository Katmanı
- Service Katmanı

- dotnet-yuzuncuyil-egitim-notları
- Çok Katmanlı Mimari Yapısı
- .NET Core ile Çok Katmanlı Yapı