Clean Architecture Mimarisinde CQRS Design Pattern(ASP.NET Core)

11 min read · 8 hours ago

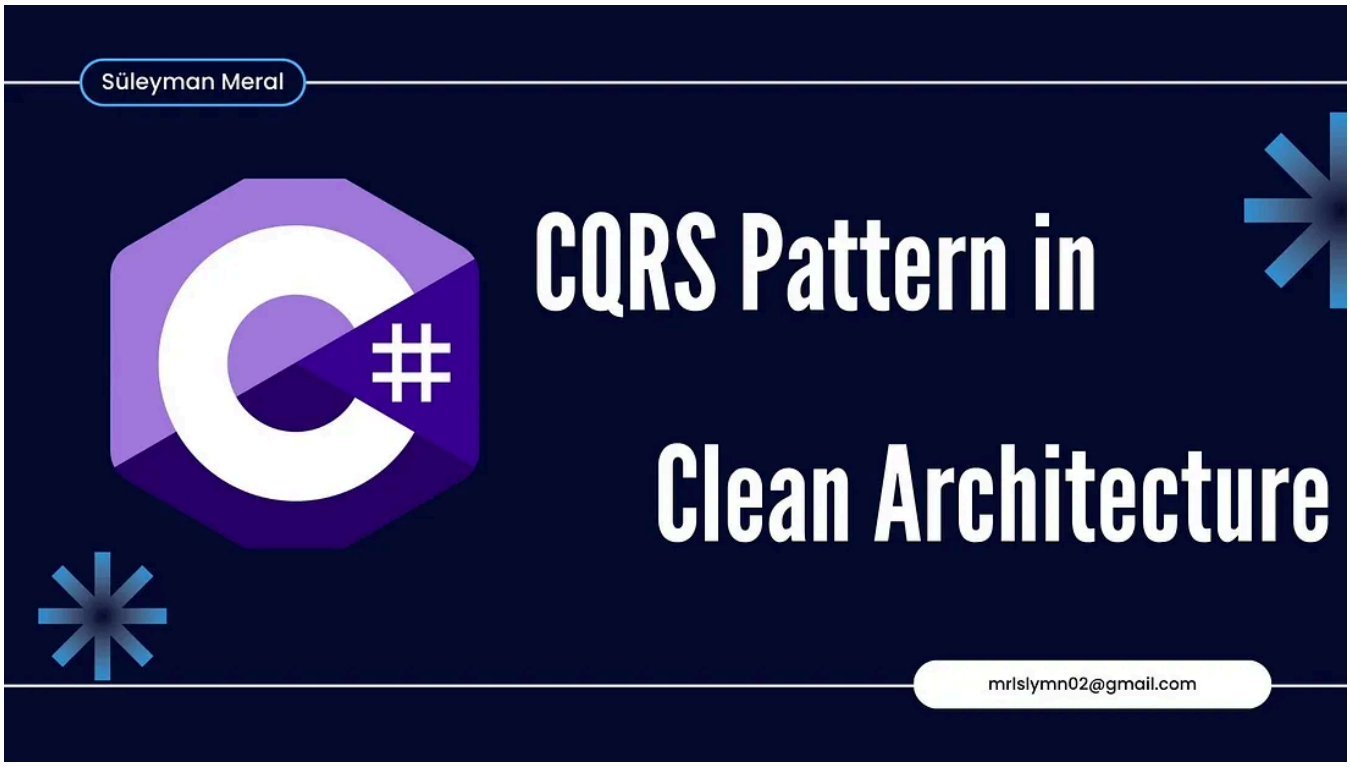
Süleyman Meral

Herkese merhaba!
Bu yazımda, Clean Architecture mimarisi üzerinden **CQRS (Command Query Responsibility Segregation)** tasarım desenini anlatmaya çalışacağım.

“CQRS tasarım deseni nedir?”, “Nasıl kullanılır?”, “Nasıl kurulur?” gibi sorulara birlikte cevap arayacağız.

Yazı biraz uzun olabilir; ancak her konuyu adım adım, sade bir dille açıklamaya çalışacağım.

Şimdiden keyifli okumalar dilerim. 😊



CQRS Design Pattern Nedir?

CQRS (Command Query Responsibility Segregation), komut ve sorgu sorumluluklarının ayrılmasını sağlayan,yazılım geliştirme sürecinde veri üzerinde yapılan işlemleri iki ayrı kategoriye ayıran bir tasarım desenidir.

Command (Komut): Sistemde bir **değişiklik** oluşturan işlemleri ifade eder. Örneğin: update,create,delete.

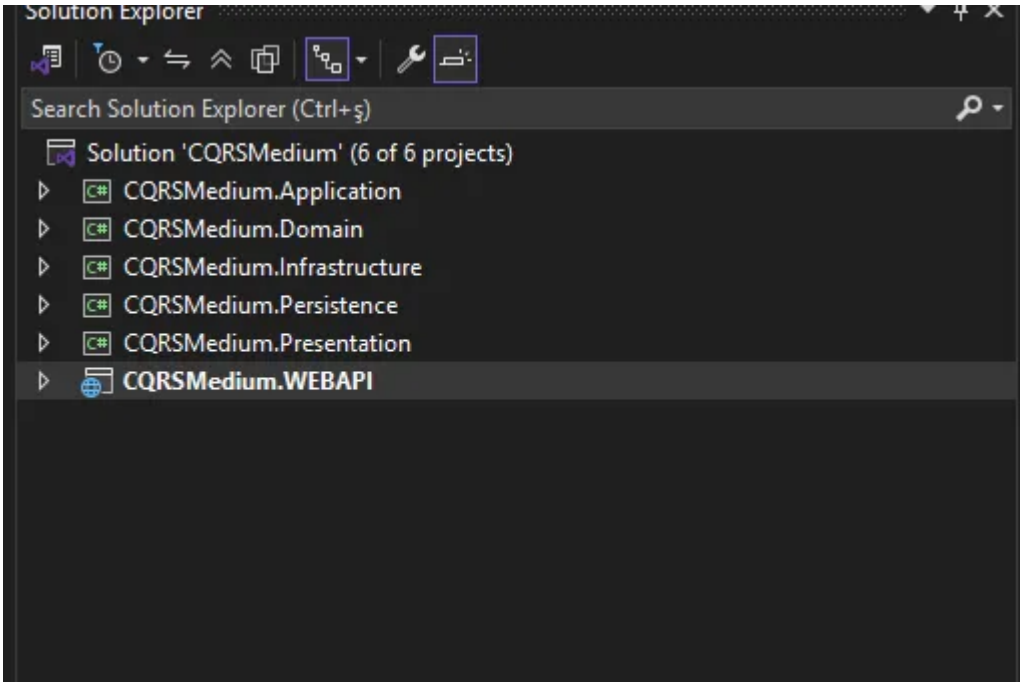
Query (Sorgu): Sistemde **yalnızca veri okuma** işlemlerini ifade eder. Örneğin: Listeleme

CQRS'in temel amacı, **okuma (Query)** ve **yazma (Command)** işlemlerinin sorumluluklarını ayırarak **uygulamanın daha esnek, ölçeklenebilir ve sürdürülebilir** olmasını sağlamaktır.

Bu desen, özellikle karmaşık sistemlerde okuma ve yazma işlemlerinin birbirinden bağımsız olarak ele alınmasına imkân tanır. Böylece her iki işlem türü için farklı optimizasyonlar, mimari çözümler ve veri modelleri kullanılabilir. Bu yapıyı clean architecture mimarisi ile birleştirirsek esnek ve ölçeklenebilir bir yapı elde ederiz.

Temel olarak CQRS tasarım desenini bu şekilde özetleyebiliriz. Teorik olarak aklımıza pek yatmamış olabilir. Şimdi ufak bir proje yaparak pratikte bunu görelim.

(Her adımı cqrs patternın dışında da ele almaya çalışacağım. Dilerseniz hızlıca geçebilirsiniz.)



Mimarimizi 6 katmandan oluşturduk. Bu katmanların ne işe yaradığını kısaca özetleyelim.

CQRSMedium.Domain:

- **Domain katmanı**, uygulamanın **çekirdek yapısını** oluşturur.
  Bu katmanda **Entity'ler**, **Value Object'ler** ve **Repository arayüzleri** gibi iş kurallarını temsil eden yapılar yer alır.
- Domain katmanı, **hiçbir dış katmana bağımlı değildir.** Bu sayede altyapı teknolojileri değişse bile (örneğin MSSQL'den PostgreSQL'e geçiş gibi), **iş kuralları etkilenmeden** geliştirilmeye devam edilebilir.
- Bu bağımsız yapı, yazılımın **esneklik**, **test edilebilirlik** ve **bakım kolaylığı** açısından güçlü olmasını sağlar.

CQRSMedium.Persistance

- Veritabanı işlemleri, repository implementasyonları burada olur.Genel olarak veritabanından sorumludur.
- Configuration işlemleri,context sınıfı burada yer alır.

CQRSMedium.Application

- Business logicleri işletir. CQRS kapsamındaki Command ve Query yapıları burada yer alır.

- Servis arayüzlerini ve implemantasyonlarını da barındırabilir.
- Ayrıca DTO ve mapping işlemleri de burada yer alabilir.

CQRSMedium.Infrastructure

- Dış sistemlerle (dosya sistemi, mail servisleri, third-party API'ler vs.) olan bağlantılar burada olur.

CQRSMedium.Presentation

- Controller yapılarının bulunduğu sunum katmanıdır.(Controllerlar direkt olarak proje tarafından görülmez. Assembly reference'ı yapmak gerektirir. Onu da gerçekleştireceğiz)

CQRSMedium.WEBAPI

- Servis kayıtları(DI),Middleware'ler,konfigürasyonlar burada yer alır.
- İlk olarak controllerlar burada yer alır. Biz onları presentation katmanına taşıyıp işleri daha da böleceğiz.

Katman yapımızı bu şekilde özetleyebiliriz. Projemizde Product entity'miz olacak ve onun üzerinden örnek yapacağız.

Domain katmanına Entities klasörü oluşturup Product classı ekliyorum.

```
public sealed class Product
{
    public Product()
    {
        Id = Guid.NewGuid().ToString();
    }

    public string Id { get; set; }
    public string Title { get; set; }=default!;
    public decimal Price { get; set; }
    public int Stock { get; set; }
}
```

Product yapımız bu şekilde.

Bu yapının veritabanı için config classını oluşturalım. Veritabanı işlemi yapacağımız için "Persistence" katmanında bu işlemi gerçekleştireceğiz.

Veritabanı olarak MSSQL ORM olarak EF Core kullancağız o yüzden EF Core ve SQL Server paketlerini “persistence” katmanına ekleyelim. Eklediğiniz paketlerin projenizde kullandığınız .NET sürümü ile aynı olmasına dikkat edin.

```
using CQRSMedium.Domain.Entities;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;


namespace CQRSMedium.Persistence.Configurations;

public sealed class ProductConfiguration : IEntityTypeConfiguration<Product>
{
    public void Configure(EntityTypeBuilder<Product> builder)
    {
        builder.Property(p => p.Price).HasColumnType("money");

    }
}
```

Product Configuration classımız bu şekilde. Configuration classlarını IEntityTypeConfiguration arayüzünden türetmemiz gerekiyor. Bu bize EF Core’un sağladığı bir özelliktir. Arayüzü implement ettiğimiz zaman Configure metodu gelecektir. builder yardımı ile çeşitli özellikler yazabiliriz. Ben decimal türünün daha sağlıklı yansıması açısından price sütununa money olacağını söyledim. HasColumnType extension metodu gözükmezse, SQL Server paketini yüklediğinizden emin olun ve tekrar yükleyin.

Entity ile işimiz şuanlık bu kadar. Şimdi ise projemizin veritabanını kuralım.

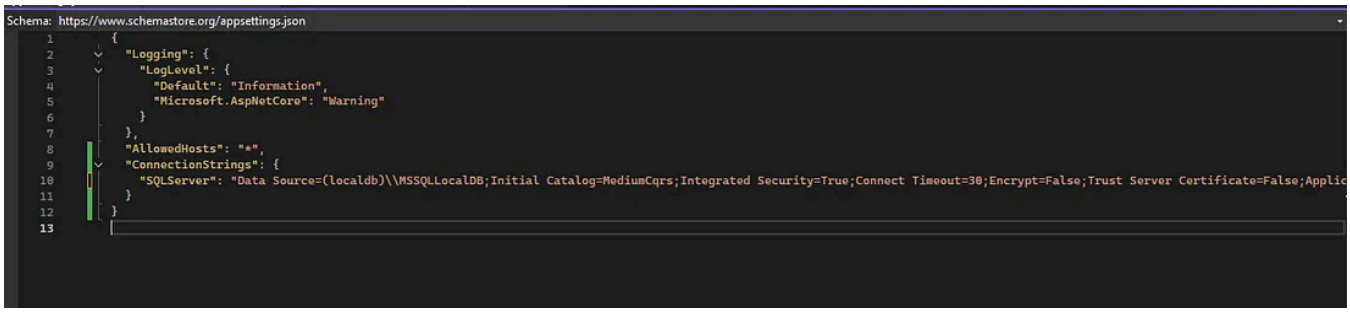
Persistence katmanına Context klasörü oluşturup AppDbContext classını ekliyorum.

```
public sealed class AppDbContext : DbContext
{

    public AppDbContext(DbContextOptions options) : base(options)
    {
    }
```

```
    public DbSet<Product> Products { get; set; }
}
```

`DbContext` sınıfına geçirilen `DbContextOptions`, Entity Framework Core'un veritabanı bağlantı bilgilerini ve yapılandırmalarını içerir.
Genellikle bu ayar, uygulama başlatılırken Dependency Injection üzerinden yapılır ve context sınıfına otomatik olarak iletilir.
Böylece `AppDbContext`, veritabanı ile sorunsuz şekilde bağlantı kurabilir.

Bu ayarı program.cs üzerinden gerçekleştireceğiz. Bundan önce veritabanı bağlantımızı oluşturalım.



appsettings.json dosyasına SqlServer şeklinde bağlantımızı ekliyoruz. SQLServer Explorer üzerinden eski bir veritabanı bağlantınızı kopyalayıp veritabanı adını değiştirebilirsiniz.

Veritabanı bağlantımıza bu dosya üzerinden erişeceğiz.

```
string connectionString= builder.Configuration.GetConnectionString("SQLServer")
builder.Services.AddDbContext<AppDbContext>(opt =>
{
    opt.UseSqlServer(connectionString);
});
```

Uygulama başlarken AppDbContext DI container'a eklenir. UseSqlServer metodu ile bağlantı stringi ayarlanır. `DbContextOptions` oluşur ve `AppDbContext`'in constructor'ına enjekte edilir. `base(options)` sayesinde `DbContext` sınıfı bağlantı bilgilerini kullanarak çalışır.

`AddDbContext<AppDbContext>(opt => opt.UseSqlServer(...))` ifadesi, EF Core'un `AppDbContext` için bir `DbContextOptions` nesnesi oluşturmasını sağlar.
Bu `options` nesnesi daha sonra `AppDbContext` constructor'ına enjekte edilir.
Bu yapı sayesinde veritabanı sağlayıcısı ve bağlantı ayarları uygulama başlatılırken merkezi olarak belirlenir.

Şimdi migration yaparak veritabanımızı code first yaklaşımı ile oluşturalım.

Bundan önce AppDbContext classımızı düzenlememiz gerekiyor.

```
using System.Reflection;

namespace CQRSMedium.Persistence;

public class AssemblyReference
{
    public static readonly Assembly assembly = typeof(AssemblyReference).Assemb
}
```

Öncelikle persistence katmanında bir AssemblyReference classı oluşturalım.

```
using CQRSMedium.Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace CQRSMedium.Persistence.Context;

public sealed class AppDbContext : DbContext
{

    public AppDbContext(DbContextOptions options) : base(options)
    {
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)=>
         modelBuilder.ApplyConfigurationsFromAssembly(typeof(AssemblyReference)

    public DbSet<Product> Products { get; set; }
}
```

Daha sonra AppDbContext classımızı güncelleyelim. Şimdi bu yapıyı bir ele alalım. Bu kodu hemen her yerde bulabilirsiniz, uygulamanız çalışır burada hiçbir problem olmaz. Ama neyi neden,nasıl yaptığınızı bilmezseniz ve bunu sorgulamayıp ezbere kod yazarsanız ileride bir hata ile karşılaşınca ortada kalırsınız. Yazılımı kod yazmaktan ibaret görmeyelim. Kod yazmaktan daha önemli bir şey varsa o da kodu okumak ve yorumlamaktır. Bu kod yazarken gelişi güzel yazalım anlamına gelmesin.

Uncle Bob'ın şu sözünü unutmayalım :

> *"The code should read like well-written prose."*
> — Robert C. Martin (Clean Code kitabı)

Şimdi konumuza dönelim.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)=>
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(AssemblyReference)
```

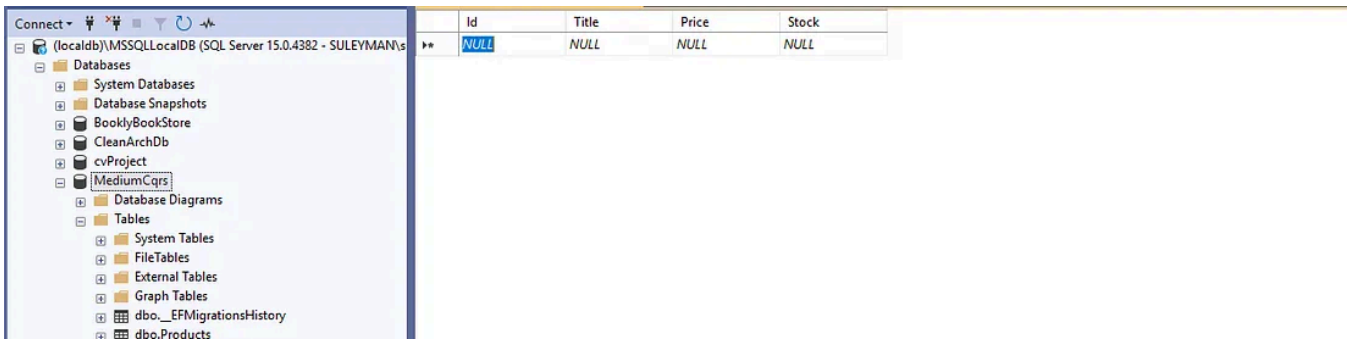
Bu metot, **Entity Framework Core'un veritabanı modelini oluştururken çağırdığı** bir metottur.
Yani EF Core'un **"modelin nasıl olacağını"** belirlediği yerdir.

```
modelBuilder.ApplyConfigurationsFromAssembly(typeof(AssemblyReference).Assembly
```

Oluşturduğumuz assembly içinde tanımlı olan tüm **IEntityTypeConfiguration<T>** implementasyonlarını bulur(Config classlarımız) ve otomatik olarak uygular.

EF Core, veritabanı tabloları ile entity sınıflarımız arasında nasıl bir eşleme yapılacağını `OnModelCreating` metodu ile belirler.
Bu metodun içinde `modelBuilder.ApplyConfigurationsFromAssembly(...)` çağrısı yaparak, tüm konfigürasyon sınıflarını (örneğin `ProductConfiguration`) otomatik olarak bulup uygulamasını sağlarız.
Bu sayede yapı daha modüler ve yönetilebilir hale gelir. Her yazdığımız config sınıfını belirtmek zorunda kalmayız.

Şimdi migration atıp veritabanımızı oluşturabiliriz.



Belirttiğimiz şekilde veritabanımız oluştu. Migration işlemini bildiğinizi varsayarak anlatmadım. Hata alırsanız Tools paketini yüklediğinizden emin olun ve işlemi Persistence katmanı üzerinden gerçekleştirin.

Entityilerimizin veritabanı işlemleri için RepositoryPatterni uygulayacağız.
Repository arayüzlerini Domain katmanında tanımlayalım.

Uygulamada veritabanı işlemlerini soyutlamak için **Repository Pattern** kullanacağız.
Repository arayüzlerini, domain kurallarını ihlal etmeden, **Domain katmanında tanımlayacağız.**
Böylece veri erişim işlemlerini dış dünyadan izole ederken, aynı zamanda test edilebilirliği ve modülerliği artıracağız.

Repository arayüzlerinin `Domain` katmanında, implementasyonlarının ise `Persistence` katmanında olması, Clean Architecture'daki **bağımlılıkların dışa doğru akması** ilkesine uygundur.
Bu yaklaşım sayesinde `Domain`, dış dünya (veritabanı, API vs.) hakkında hiçbir şey bilmeden çalışabilir.

Repository Pattern, veritabanı işlemlerini uygulamanın geri kalanından ayırmak için kullanılan bir design patterndir.
Bu pattern sayesinde uygulama, veritabanının nasıl çalıştığını bilmeden sadece tanımlanan arayüzler (interface'ler) üzerinden veri ile iletişim kurar.

```
using CQRSMedium.Domain.Entities;

namespace CQRSMedium.Domain.Repositories;
```

```
    public interface IProductRepository
    {
        Task AddAsync(Product product);
        Task<List<Product>> GetAllAsync();
    }
```

Repository methodlarını `async` olarak tanımlıyoruz çünkü veritabanı işlemleri genellikle IO-bound (giriş/çıkış odaklı) işlemlerdir.
Bu tür işlemleri asenkron şekilde yazmak, uygulamanın kaynaklarını daha verimli kullanmasını ve aynı anda çok sayıda isteği karşılayabilmesini sağlar.
Ayrıca modern veri erişim araçları (EF Core, Dapper) zaten asenkron methodlar sunmaktadır, bu yüzden `await` ile çağrılmaları gerekir.

Bu repository'nin implementasyonunu persistence katmanında yapalım.

```
using CQRSMedium.Domain.Entities;
using CQRSMedium.Domain.Repositories;
using CQRSMedium.Persistence.Context;
using Microsoft.EntityFrameworkCore;

namespace CQRSMedium.Persistence.Repositories;

public sealed class ProductRepository:IProductRepository
{
    private readonly AppDbContext _context;

    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task AddAsync(Product product)
    {
       await _context.Products.AddAsync(product);
       await _context.SaveChangesAsync();
    }

    public async Task<List<Product?>> GetAllAsync()
    {
        return await _context.Products.ToListAsync();
```

```
    }
}
```

Repository katmanında `AppDbContext`'i constructor üzerinden enjekte ederek `_context` nesnesi aracılığıyla veritabanı işlemlerimizi gerçekleştiriyoruz.
Bu işlemleri, Entity Framework Core'un sunduğu altyapı sayesinde kolaylıkla yapabiliyoruz.

Bu örnekte yalnızca bir adet `Product` entity'si ile çalışıyor olsak da, gerçek dünya projelerinde onlarca hatta yüzlerce entity yer alabilir.
Her entity için ayrı ayrı repository sınıfı yazmak ve aynı CRUD işlemlerini defalarca tekrar etmek hem zaman kaybı olur hem de kod tekrarına yol açar.

Bu problemi çözmek için genellikle **generic repository yapısı** tercih edilir.
Böylece temel veri işlemleri (`Add`, `Update`, `Delete`, `GetAll`, `GetById` gibi) bir `BaseRepository<T>` sınıfında tanımlanır ve diğer repository'ler bu sınıftan türetilir.

Dilerseniz bu yapıyı kendiniz oluşturabilir veya **Ardalis.Specification**, **SharpRepository** gibi topluluk tarafından geliştirilmiş ve sıkça kullanılan hazır çözümlerden de faydalanabilirsiniz.

Şimdi asıl konumuz olan CQRS pattern'i uygulamaya başlayalım.

CQRS pattern'i uygularken en popüler kütüphane olan MediatR kütüphanesini kullanacağız. Bu kütüphaneyi Application,WEBAPI,Presentation katmanlarına kurmamız gerekiyor.

Kurulum işleminden sonra WEBAPI katmanında bu servisin kaydını yapmamız gerekli.

```
builder.Services.AddMediatR(cfr =>
{
    cfr.RegisterServicesFromAssembly( );
});
```

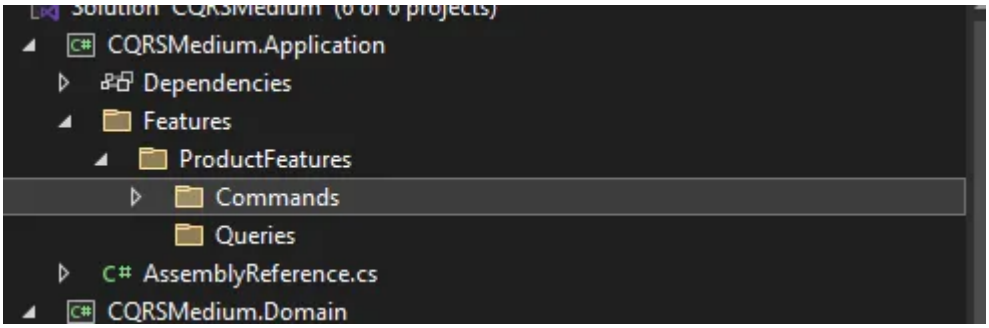
Kayıt yaparken bizden bir assemblyreference istiyor. Bu referansı MediatR uyguladığımız katmandan almamız gerekiyor. Bu da Application katmanıdır.

Persistence katmanında yaptığımız gibi bir AssemblyReference classı oluşturup buraya verelim.

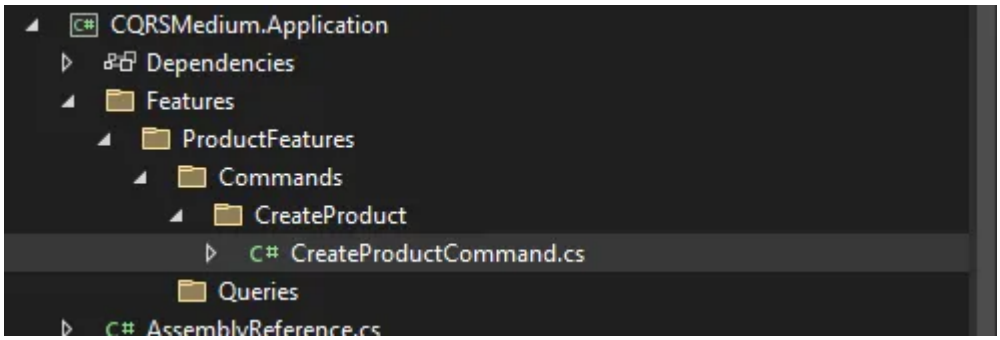
```
builder.Services.AddMediatR(cfr =>
{
    cfr.RegisterServicesFromAssembly(typeof(CQRSMedium.Application.AssemblyRefe
});
```

Application katmanını görmez ise proje referansı olarak eklerseniz düzelecektir.

Servis kaydımız tamam. Şimdi Application katmanına geri dönelim.



Features adında bir klasör oluşturdum. Entitylere ait işlemler burada yer alacak. ProductFeateures commands ve queries olarak ayırlmış durumda. Bu sayede istediğimiz yapıyı elde edeceğiz. İlk olarak ekleme işlemi için commands klasörünün altına CreateProduct adında bir klasör oluşturuyorum.



CreateProduct klasörüne CreateProductCommand adında bir class ekledim. Bu Controllerdan istenen requeste parametre olarak ne vereceksek onu içerecektir.

```
using CQRSMedium.Application.Features.ProductFeatures.Commands.CreateProduct;
using MediatR;


namespace CQRSMedium.Application.Features.ProductFeatures.Commands.CreateProduc

public sealed record CreateProductCommand(
    string Title,
    decimal Price,
    int Stock) : IRequest<CreateProductCommandResponse>; // IRequest arayüzü, k
```

`CreateProductCommand` sınıfı, uygulamada bir ürün oluşturma işlemini temsil eden bir **komut (Command)** nesnesidir. Bu komut, daha sonra yazılacak olan `CreateProductCommandHandler` sınıfı tarafından işlenecektir. `CreateProductCommand`, MediatR kütüphanesinden gelen `IRequest<TResponse>` arayüzünü implemente eder ve bu sayede işlem sonucunda `CreateProductCommandResponse` türünde bir yanıt döneceğini belirtir.

Bir istek `Controller` üzerinden gönderildiğinde, MediatR çalışma zamanında (`runtime`) `CreateProductCommand` tipinde bir `IRequest` nesnesinin iletildiğini algılar. Ardından, bu komutu işleyebilecek uygun `IRequestHandler<CreateProductCommand, CreateProductCommandResponse>` arayüzünü implement eden handler sınıfını, yani `CreateProductCommandHandler`'ı bulur.

MediatR, bu eşleştirmeyi arka planda otomatik olarak gerçekleştirir. Handler bulunduğunda, `Handle` metodu çağrılır ve `CreateProductCommand` parametresi bu metoda iletilir. Doğru arayüzlerin kullanılması, bu sürecin sorunsuz ve etkin bir şekilde işlemesini sağlar.

```
namespace CQRSMedium.Application.Features.ProductFeatures.Commands.CreateProduc

public sealed record CreateProductCommandResponse(
    string message= "Product created successfully."

    );
```

CreateProductCommandResponse classını da bu şekilde tanımladık. Şimdi sırada Commandi işleyecek Handler sınıfını yazmak var.

```
public sealed class CreateProductCommandHandler : IRequestHandler<CreateProduct
{
    public Task<CreateProductCommandResponse> Handle(CreateProductCommand reque
    {
        throw new NotImplementedException();
    }
}
```

CreateProductCommandHandler classımız IRequestHandler arayüzünden türeyecek. Bu arayüze Command classımızı ve response classımızı veriyoruz. Bu arayüz de MediatR kütüphanesinden gelen bir arayüzdür. İşlemlerimizi gerçekleştireceğimiz asıl yer burası.

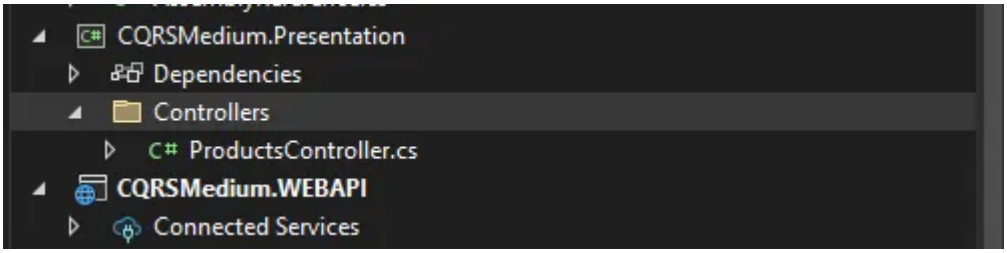
Bir servis oluşturup ekleme işlemi burada yapılabilir. Fakat uygulamamız küçük olduğu için ben direkt repository üzerinden işlemi gerçekleştireceğim.

```
using CQRSMedium.Application.Features.ProductFeatures.Commands.CreateProduct;
using CQRSMedium.Application.Features.ProductFeatures.Commands.CreateProductCom
using CQRSMedium.Domain.Entities;
using CQRSMedium.Domain.Repositories;
using MediatR;

public sealed class CreateProductCommandHandler : IRequestHandler<CreateProduct
{
    private readonly IProductRepository _productRepository;

    public CreateProductCommandHandler(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public async Task<CreateProductCommandResponse> Handle(CreateProductCommanc
    {
        await _productRepository.AddAsync(new Product
        {
            Title = request.Title,
            Price = request.Price,
            Stock = request.Stock
```

```
            });

            return new();
        }
    }
```

Create işlemini bu şekilde gerçekleştirebiliriz. Product newlemek çok doğru bir yaklaşım değil burada mapping işlemi uygulamak daha doğru bir yaklaşımdır. Fakat dediğim gibi proje küçük olduğu için bazı şeyleri gözardı ediyoruz. Amacımız cqrs mantığını clean arch çerçevesinde anlamak. Dilerseniz daha detaylı bir projeye yazının sonunda vereceğim github hesabımdan bakabilirisiniz.

Handle sınıfımızı da yazdık. Şimdi Controller oluşturup test edelim. (Validasyon işlemi yapmadık. Fluent Validation ile kolay bir şekilde validator yazabilirsiniz. Bu validator classını CreateProductCommandValidator olarak CreateProduct klasörüne ekleyebilirsiniz.



Presentation katmanına Controllers klasörü oluşturuyorum. İçerisine ProductsController adında bir class ekliyorum. Fakat şöyle bir sorunumuz var uygulama çalıştığı zaman projemiz bu controller classını görmez . Bunun için bir ayar yapmamız gerekiyor. Daha önce gerçekleştirdiğimiz gibi bu katmanında AssemblyReference classını oluşturalım.

Diğer katmanlardaki assembly reference classını kopyalayıp yapıştırmış olabilirsiniz. Lütfen namespace'inizi değiştirin. Yoksa hata alırsınız.

AssemblyReference classını oluşturduktan sonra program.cs e gelelim.

```
builder.Services.AddControllers()
            .AddApplicationPart(typeof(
```

```
            CQRSMedium.Presentation.AssemblyReference).Assembly);  //controllera
```

AssemblyReference classını buraya vermemiz gerekiyor. Bu yaptığımız işlemlerin temelinde Reflection kavramı vardır. O konu hakkında araştırma yapmanızı kesinlikle öneririm. Daha önce bu konu ile alakalı bir örnek yazı paylaşmıştım. Dilerseniz ona da bakabilirsiniz.

.NET CORE Reflection(Runtime'da Kodu Okumak ve Değiştirmek: Reflection'ın Gücü)

Herkese merhaba! Bu yazımda sizlere Reflection kavramından bahsedeceğim. Reflection Nedir? Niçin Kullanılır...

medium.com

Referans işlemimiz tamamlandı. Şimdi Controller yazmaya devam edebiliriz.

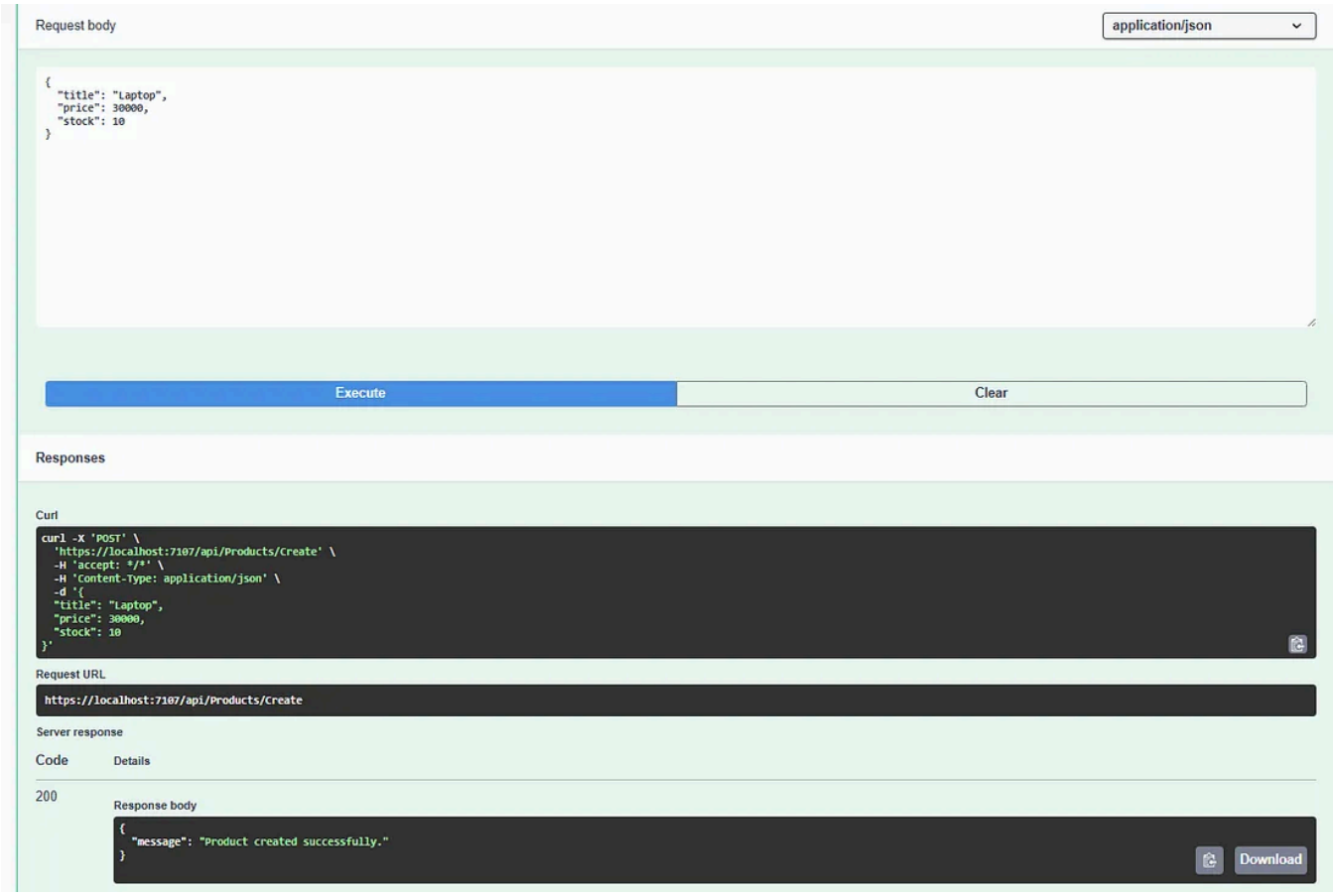
```
namespace CQRSMedium.Presentation.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        public readonly IMediator _mediator;

        public ProductsController(IMediator mediator)
        {
            _mediator = mediator;
        }
        [HttpPost("[action]")]
        public async Task<IActionResult> Create(CreateProductCommand request, C
        {
            CreateProductCommandResponse response = await _mediator.Send(reques
            return Ok(response);
        }
    }
}
```

CreateProductCommand'i request olarak meditor ile gönderiyoruz. Handler sınıfı bu requesti yakalıyor ve işliyor ve response olarak bir değer döndürüyor. Uygulamamız test için hazır. Fakar Repository DI kaydını yapmayı unuttuk. Onu da gerçekleştirelim.

```
builder.Services.AddScoped<IProductRepository, ProductRepository>();
```

Program.cs üzerinden kaydı gerçekleştirebiliriz.

Şimdi uygulamayı başlatıp swagger üzerinden test yapalım.



Sorunsuz bir şekilde işlemimiz gerçekleşti. Peki nasıl oldu? sırayla adım adım açıklayalım. Daha sonra query'mizi yazarız.

```
        }
        [HttpPost("[action]")]
        public async Task<IActionResult> Create(CreateProductCommand request, Cance
        {
```

```
        CreateProductCommandResponse response = await _mediator.Send(request, c
        return Ok(response);
    }
```

Swagger ile tetiklediğimiz yer burası. Create metodu request olarak CreateProductCommand alıyor. Bu bizim swagger da gördüğümüz response body. Burada oluşan CreateProductCommand nesnesi mediator ile Send ediliyor. Send edilen bu nesne CreateProductCommandHandler sınıfı tarafından yakalanıyor ve işleniyor.

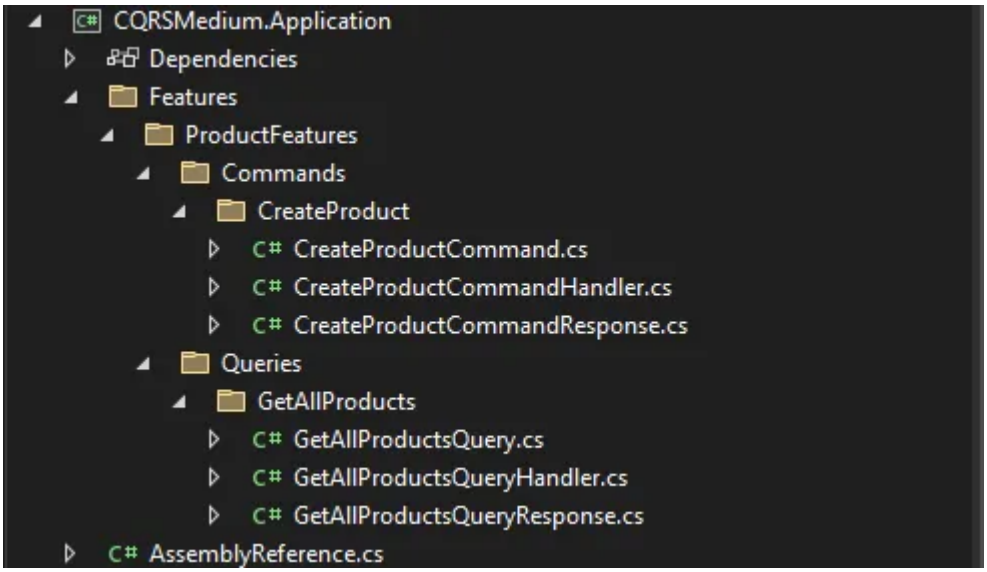
```
    public async Task<CreateProductCommandResponse> Handle(CreateProductCommanc
    {
        await _productRepository.AddAsync(new Product
        {
            Title = request.Title,
            Price = request.Price,
            Stock = request.Stock
        });

        return new();
    }
```

IRequestHandlerin implemen ettiği bu metod kullanılıyor. Bu metod bize bir response döndürüyor. Bu response bizim CreateProductCommand de verdiğimiz CreateProductCommandResponse'a karşılık gelmektedir.

```
CreateProductCommandResponse response = await _mediator.Send(request, cancellat
```

Oradan dönen değer response değişkenine atanıp return ediliyor. Ve işlem başarılı bir şekilde gerçekleşiyor. Daha iyi kavramak amacı ile bir breakpoint kullanıp uygulamayı debug modda izleyebilirsiniz.

Şimdi ise query işlemimizi yapalım.




```
public record GetAllProductsQuery():IRequest<List<GetAllProductsQueryResponse>>
```

Bu querynin bize GetAllProductsQueryResponse türünde bir response döneceğini söylüyorum.

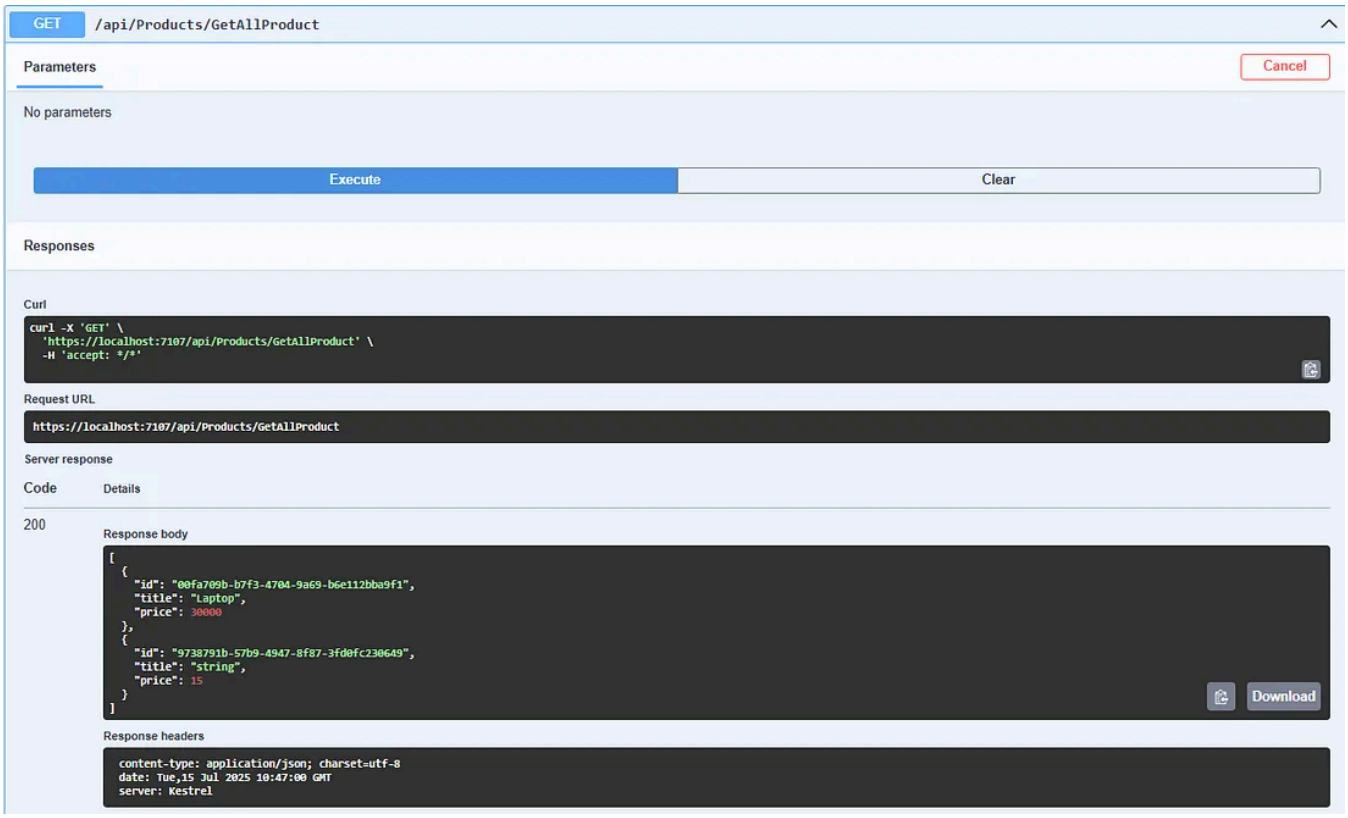
```
public record GetAllProductsQueryResponse(
    string Id,
    string Title,
    decimal Price
    );
```

```
public sealed class GetAllProductsQueryHandler : IRequestHandler<GetAllProduc
{
    private readonly IProductRepository _productRepository;

    public GetAllProductsQueryHandler(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public async Task<List<GetAllProductsQueryResponse>> Handle(GetAllProduct
    {
        var products = await _productRepository.GetAllAsync();

        var response = products.Select(p => new GetAllProductsQueryResponse(
            p.Id.ToString(),
```

```
                p.Title,
                p.Price
            )).ToList();

            return response;
        }
    }
```

Şimdi controllerda bir action yazıp test edelim.

```
namespace CQRSMedium.Presentation.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        public readonly IMediator _mediator;

        public ProductsController(IMediator mediator)
        {
            _mediator = mediator;
        }
        [HttpPost("[action]")]
        public async Task<IActionResult> Create(CreateProductCommand request, C
        {
            CreateProductCommandResponse response = await _mediator.Send(reques
            return Ok(response);
        }
        [HttpGet("[action]")]
        public async Task<IActionResult> GetAllProduct(CancellationToken cancel
        {
            var query=new GetAllProductsQuery();
            var result= await _mediator.Send(query, cancellationToken);
            return Ok(result);
        }
    }
}
```



Bu işlem de başarılı bir şekilde gerçekleşti.

Ayrıca eklemek gerekir ki, `CancellationToken` yapısı, client tarafında bir işlem tamamlanmadan önce iptal edilmesine olanak tanır. Bu sayede uzun sürebilecek işlemler sırasında kaynak kullanımı optimize edilir ve istemci taleplerine daha hızlı yanıt verilebilir. Projede `CancellationToken` desteği, MediatR üzerinden iletilen işlemlerde iptal mekanizmasını etkin kılmak amacıyla kullanılmıştır.

CQRS (Command Query Responsibility Segregation) deseni sayesinde uygulamadaki görevleri sorgular ve komutlar olarak ayırarak, **daha okunabilir**, **bakımı kolay** ve **geliştirilebilir** bir yapı kurduk. Bu yapıyı geliştirirken, Clean Architecture prensiplerine uygun bir katmanlı mimari oluşturmaya özen gösterdik.

Projenin ölçeği küçük olduğundan, bazı durumlarda Clean Code prensiplerinden bilinçli olarak sapmalar yapılmış olabilir (örneğin: servis katmanı oluşturmamak, mapping işlemlerini manuel gerçekleştirmek gibi). Ancak bu tercihlerin her biri, kod içerisinde açıklamalarla ifade edilerek gerekçelendirilmiştir.

Bu sayede proje, hem mimari prensiplere uygun hem de yalın bir yapıda kalacak şekilde planlanmıştır.

Yazılım | Cqrs Pattern | Clean Architecture

Edit profile

Written by Süleyman Meral

6 followers · 7 following

Backend Developer(.NET Core)

No responses yet

Süleyman Meral

What are your thoughts?

More from Süleyman Meral