Data Shaping Nedir ? (ASP.NET Core)

8 min read · Jun 10, 2025

Süleyman Meral

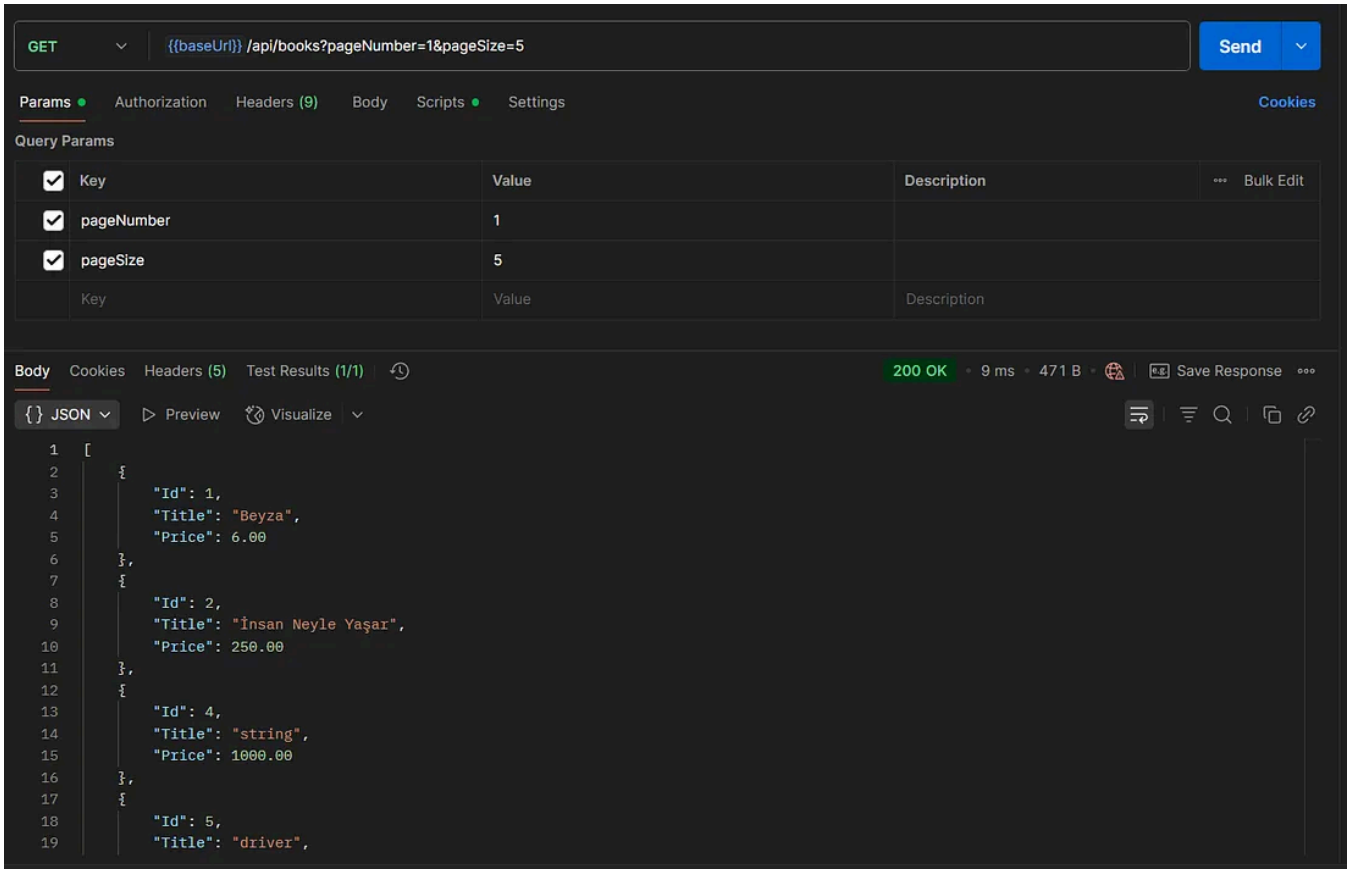
Herkese merhaba. Bu yazımda sizlere API projelerimizde kullanabileceğimiz bir işlem olan “Data Shaping” işleminden bahsedeceğim. Data Shaping Nedir? Nasıl Kullanılır? Ne İşe Yarar ? gibi sorulara yanıt vereceğiz. Ve bir API projesi üzerinden konuyu pekiştireceğiz. Şimdiden keyifli okumalar dilerim.

Data Shaping Nedir?

Data Shaping, istemciden gelen taleplere göre yalnızca ihtiyaç duyulan filed’ların seçilerek sunulmasını sağlayan bir tekniktir. Özellikle RESTful API’lerde, istemcinin(client) gereksiz verileri almasını engelleyerek veri trafiğini azaltır, performansı artırır ve güvenlik açısından daha kontrollü bir yapı sunar. Bu yöntem,

istemcinin örneğin sadece `Name` ve `Price` alanlarını talep etmesi durumunda, sunucunun sadece bu alanları içeren nesneleri döndürmesiyle uygulanır. C# tarafında `ExpandoObject` gibi dinamik yapılarla birlikte reflection kullanılarak esnek bir biçimde gerçekleştirilebilir.

Örnek verecek olursak kitap listesi döndüren bir API endpointimiz olsun.

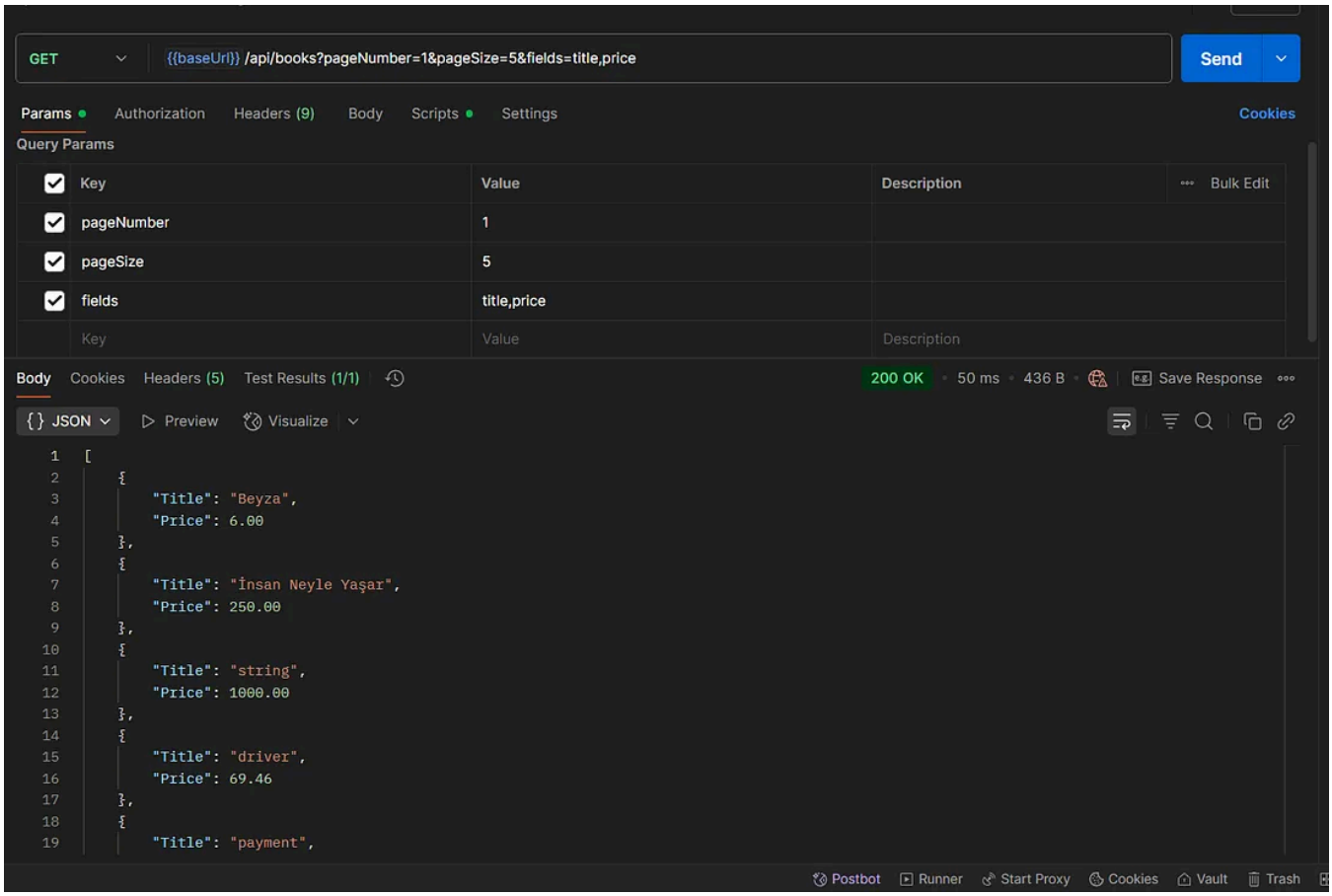


API'mizin get isteği aslında Book entity'mizin property'lerini döndürüyor.

```
public class Book
{
    public int Id { get; set; }
    public string? Title { get; set; }
    public decimal Price { get; set; }
}
```

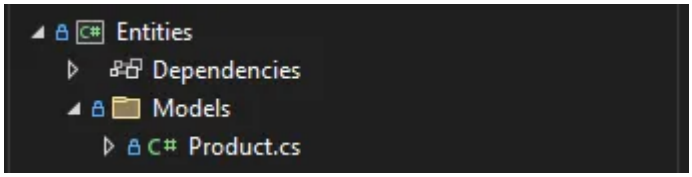
Bu örnekte sınırlı sayıda property'miz olduğu için fark net görünmeyebilir. Ancak gerçek dünyada, istemci tarafının yalnızca belirli alanlara ihtiyaç duyduğu durumlarla sıkça karşılaşırız. Örneğin, frontend geliştiricileri bir ürün listesi için yalnızca `Title` ve `Price` alanlarının döndürüldüğü bir endpoint talep edebilir. Böyle

bir senaryoda, API'nin bu alanları filtreleyerek sadece istenilen bilgileri sağlaması hem performans hem de veri yönetimi açısından önemlidir. Bu ihtiyacı karşılamak için **Data Shaping** tekniği kullanılır. Alternatif olarak, **Specification Pattern** ile birlikte **Projection** teknikleri de tercih edilebilir. Ancak Data Shaping, dinamik alan seçimi konusunda daha esnek bir çözüm sunar.



Yukarıda da görüldüğü üzere API sadece istenilen field'ları döndürdü. Şimdi bu işlemi nasıl gerçekleştireceğiz onu inceleyelim.

Bir ASP.NET Core Web API projesi oluşturalım. Daha sonra içerisine Entites katmanını ekleyelim.



Örnek olarak Product entity'si üzerinden çalışacağız.

```
public class Product
{
    public int Id { get; set; }
    public required string Name { get; set; }

    public required string Brand { get; set; }
    public required string Description { get; set; }
    public decimal Price { get; set; }
    public required string PictureUrl { get; set; }
    public required string Type { get; set; }
    public int QuantityInStock { get; set; }
}
```

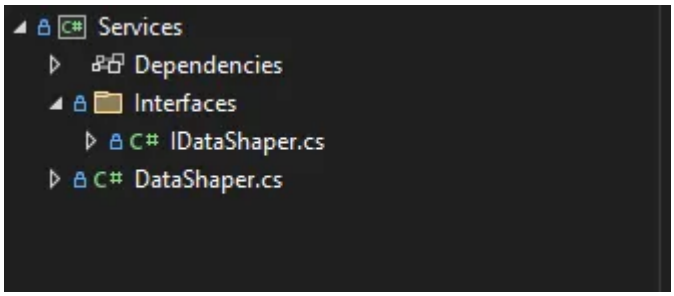
Product Entity'miz yukarıda görülen property'leri içinde barındırıyor. Bu entity'yi tüketirsek bize bütün alanlar dönecektir. Fakat biz sadece istenilen alanların dönmesini istiyoruz. Bunun içinde farklı bir endpoint tasarımı gerçekleştireceğiz.

Konuyu çok uzatmamak adına veri tabanına bağlanmak verine mock data ile çalışacağız. Bunun için MockData adında bir klasör oluşturup içerisine ProductStore adında bir class ekliyoruz.

```
public static class ProductStore
{
    public static List<Product> Products = new List<Product>
{
    new Product
    {
        Id = 1,
        Name = "Wireless Mouse",
        Brand = "Logitech",
        Description = "Ergonomic wireless mouse with fast scrolling and siler
        Price = 29.99m,
        PictureUrl = "https://example.com/images/wireless-mouse.jpg",
        Type = "Electronics",
        QuantityInStock = 50
    },
    new Product
    {
        Id = 2,
        Name = "Mechanical Keyboard",
        Brand = "Keychron",
        Description = "RGB backlit mechanical keyboard with red switches.",
        Price = 89.99m,
        PictureUrl = "https://example.com/images/mechanical-keyboard.jpg",
        Type = "Electronics",
```

```
            QuantityInStock = 25
        },
        new Product
        {
            Id = 3,
            Name = "Running Shoes",
            Brand = "Nike",
            Description = "Lightweight running shoes with breathable mesh fabric.
            Price = 59.99m,
            PictureUrl = "https://example.com/images/running-shoes.jpg",
            Type = "Footwear",
            QuantityInStock = 100
        },
        new Product
        {
            Id = 4,
            Name = "Smartwatch",
            Brand = "Apple",
            Description = "Apple Watch Series 8 with GPS and heart-rate tracking.
            Price = 399.00m,
            PictureUrl = "https://example.com/images/smartwatch.jpg",
            Type = "Wearables",
            QuantityInStock = 15
        },
        new Product
        {
            Id = 5,
            Name = "Coffee Maker",
            Brand = "Philips",
            Description = "Automatic drip coffee maker with timer and strength cc
            Price = 49.99m,
            PictureUrl = "https://example.com/images/coffee-maker.jpg",
            Type = "Home Appliances",
            QuantityInStock = 30
        }
    };
    }
```

MockData işlemimiz tamamlandı. Şimdi Data Shaping işlemini gerçekleştireceğimiz Services katmanını oluşturalım. Ve içerisine IDataShaper interface'ini ve DataShaper classını ekleyelim.




```
public interface IDataShaper<T>
{
    IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string fie
}
```

`ExpandoObject`, çalışma zamanında (runtime) içerisine dinamik olarak property ekleyebildiğimiz bir sınıftır. Bu, hangi alanların döneceği önceden belli değilse çok kullanışlıdır — işte burada **Data Shaping** devreye girer.

```
public class DataShaper<T> : IDataShaper<T>
{
    public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, st
    {
        var propertyInfos = typeof(T).GetProperties(BindingFlags.Public | B
        var fieldList = string.IsNullOrWhiteSpace(fields)
            ? propertyInfos.Select(p => p.Name)
            : fields.Split(',', StringSplitOptions.RemoveEmptyEntries).Sele

        var shapedData = new List<ExpandoObject>();

        foreach (var entity in entities)
        {
            var shapedObject = new ExpandoObject() as IDictionary<string, c

            foreach (var field in fieldList)
            {
                var prop = propertyInfos.FirstOrDefault(p =>
                    p.Name.Equals(field, StringComparison.InvariantCultureI

                if (prop != null)
                {
                    shapedObject[field] = prop.GetValue(entity);
                }
            }
```

```
                shapedData.Add((ExpandoObject)shapedObject);
            }

            return shapedData;
        }
```

DataShaper classımız IDataShaper arayüzünden türedi. Ve arayüzde imzaladığımız metodu buraya implement ettik. Şimdi metodumuzun içeriğini inceleyelim.

```
    public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string fie
```

`ShapeData` metodu, iki parametre alır: `entities` ve `fields`.

`entities` parametresi, genelleştirilmiş (`generic`) bir koleksiyon olup dinamik olarak şekillendirilecek varlıkları (örneğin: `Product`, `Book` gibi) temsil eder.

`fields` parametresi ise, istemciden (örneğin bir API çağrısından) gelen ve yanıt içerisinde yer alması istenen alanları virgülle ayrılmış bir biçimde ifade eder.

GET {{baseUrl}}/api/books?pageNumber=1&pageSize=5&fields=title,price

`fields` parametresine `"title,price"` değerini iletir. Bu durumda `ShapeData` metodu, her bir varlık nesnesinden yalnızca `Title` ve `Price` alanlarını alır, diğer property'leri dışarıda bırakır. Bu işlem sonucunda `ExpandoObject` türünden dinamik nesneler koleksiyonu elde edilir ve yalnızca ihtiyaç duyulan alanların istemciye döndürülmesi sağlanır. Bu yaklaşım, veri taşımayı optimize eder ve API performansını artırır.

```
    var propertyInfos = typeof(T).GetProperties(BindingFlags.Public | BindingFlags.
```

`propertyInfos` değişkeni, .NET reflection API'si kullanılarak belirtilen `T` tipinin public instance property'lerine erişim sağlar. `typeof(T).GetProperties()` metodu,

`BindingFlags.Public` ve `BindingFlags.Instance` parametreleriyle filtrelenerek yalnızca:

- Public erişim seviyesine sahip (ör: `public string Name { get; set;}`),
- Static olmayan (nesne örneğine bağlı)
  property'lerin `PropertyInfo` nesnelerini döndürür.

```
var fieldList = string.IsNullOrWhiteSpace(fields)
                ? propertyInfos.Select(p => p.Name)
                : fields.Split(',', StringSplitOptions.RemoveEmptyEntries).Sele
```

`fieldList` Değişkeninin İşlevi
Bu kod parçası, dinamik alan seçimi (field projection) için tasarlanmıştır.

- `fields` parametresinin boş (`null` / `whitespace`) olup olmadığını kontrol eder.
- Boşsa, tüm public property'leri (`propertyInfos.Select(p => p.Name)`) seçer.
- Boş değilse, `fields` string'ini virgülle ayırır (`Split`),
- Boşlukları temizler (`Trim`),
- Boş elemanları atar (`RemoveEmptyEntries`).

```
"title, price,  " → ["title", "price"]  // Boşluklar ve gereksiz virgüller filt
```

```
var shapedData = new List<ExpandoObject>();
```

- `List<ExpandoObject>` tipinde tanımlanan `shapedData` , runtime'da dinamik olarak genişletilebilen nesneler koleksiyonudur

```
foreach (var entity in entities)
            {
                var shapedObject = new ExpandoObject() as IDictionary<string, c

                foreach (var field in fieldList)
                {
                    var prop = propertyInfos.FirstOrDefault(p =>
                        p.Name.Equals(field, StringComparison.InvariantCultureI

                    if (prop != null)
                    {
                        shapedObject[field] = prop.GetValue(entity);
                    }
                }

                shapedData.Add((ExpandoObject)shapedObject);
            }
```

Foreach döngüsü ile entity nesneleri üzerinde dönüyoruz.

```
var shapedObject = new ExpandoObject() as IDictionary<string, object>;
```

ExpandoObject nesnesini IDictionary türüne çeviriyoruz. Bunun temel sebebi field ismi bir anahtar olarak kullanılabilir,bu da işimizi kolaylaştırır.

```
foreach (var field in fieldList)
```

Listedeki her field üzerinde dönüyoruz.

```
var prop = propertyInfos.FirstOrDefault(p =>
    p.Name.Equals(field, StringComparison.InvariantCultureIgnoreCase));
```

Bu satır, `field` ismine karşılık gelen property'yi `propertyInfos` listesinden bulur. Büyük-küçük harf duyarsız eşleştirme yapar.

```
if (prop != null)
{
    shapedObject[field] = prop.GetValue(entity);
}
```

Eğer ilgili field için (örneğin:price) bir property bulunmuşsa (product entity'sine ait price property'si olabilir) bu property'nin değerini alır. Daha sonra shapedObject sözlüğüne ekler.

```
shapedData.Add((ExpandoObject)shapedObject);
```

Tüm istenen alanları içeren `shapedObject`, `ExpandoObject` olarak `shapedData` isimli listeye eklenir.

```
return shapedData;
```

Son olarak bu liste döndürülür.

DataShaper metodumuzu yazdık. Şimdi bu yapıyı controller üzerinde kullanalım ve yapıyı test edelim.

```
        private readonly IDataShaper<Product> _dataShaper;

        public ProductController(IDataShaper<Product> dataShaper)
        {
            _dataShaper = dataShaper;
        }

        [HttpGet]
        public IActionResult GetBooks([FromQuery] string? fields)
```

```
        {
            var products= ProductStore.Products;
            var shapedData = _dataShaper.ShapeData(products, fields ?? "");
            return Ok(shapedData);
        }
```

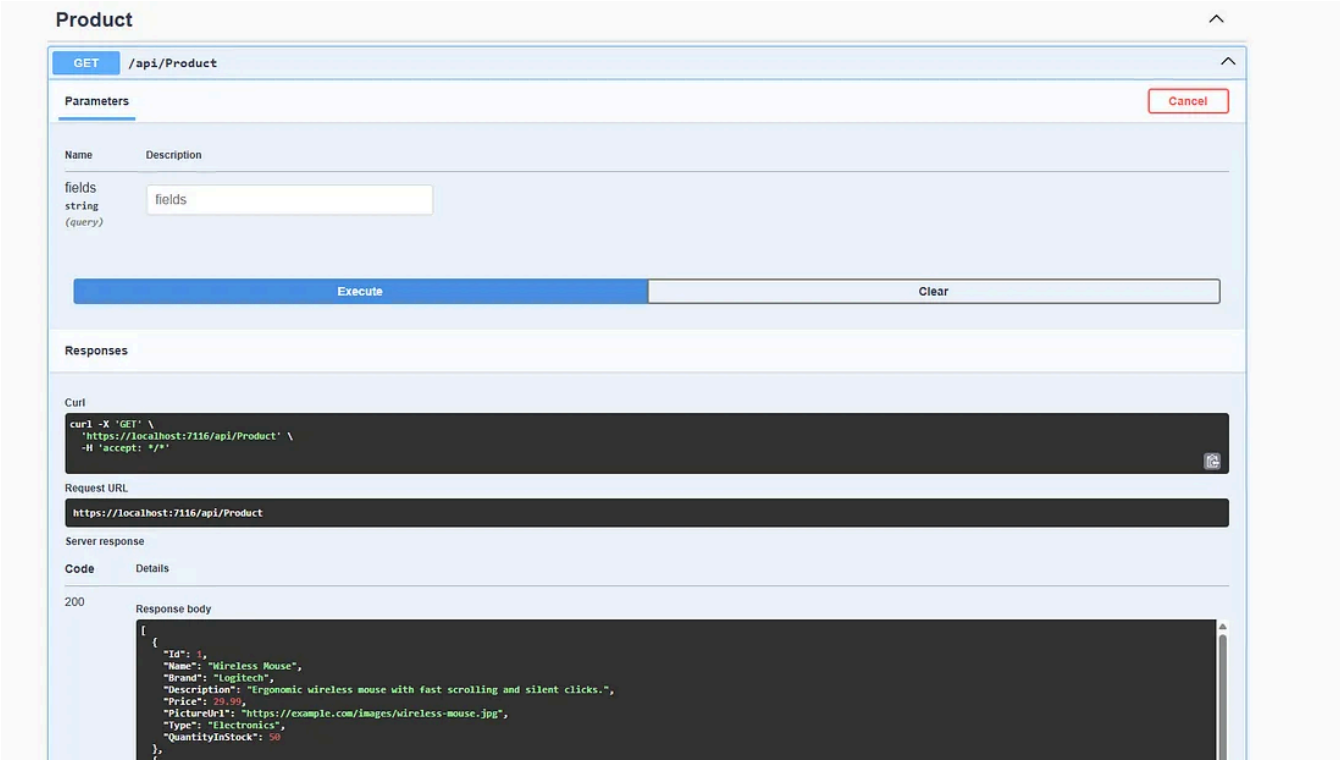
(Servisimizin kaydını program.cs üzerinden yapmayı unutmayalım.)

`fields` URL'den gelen bir query parametresi. Örneğin:

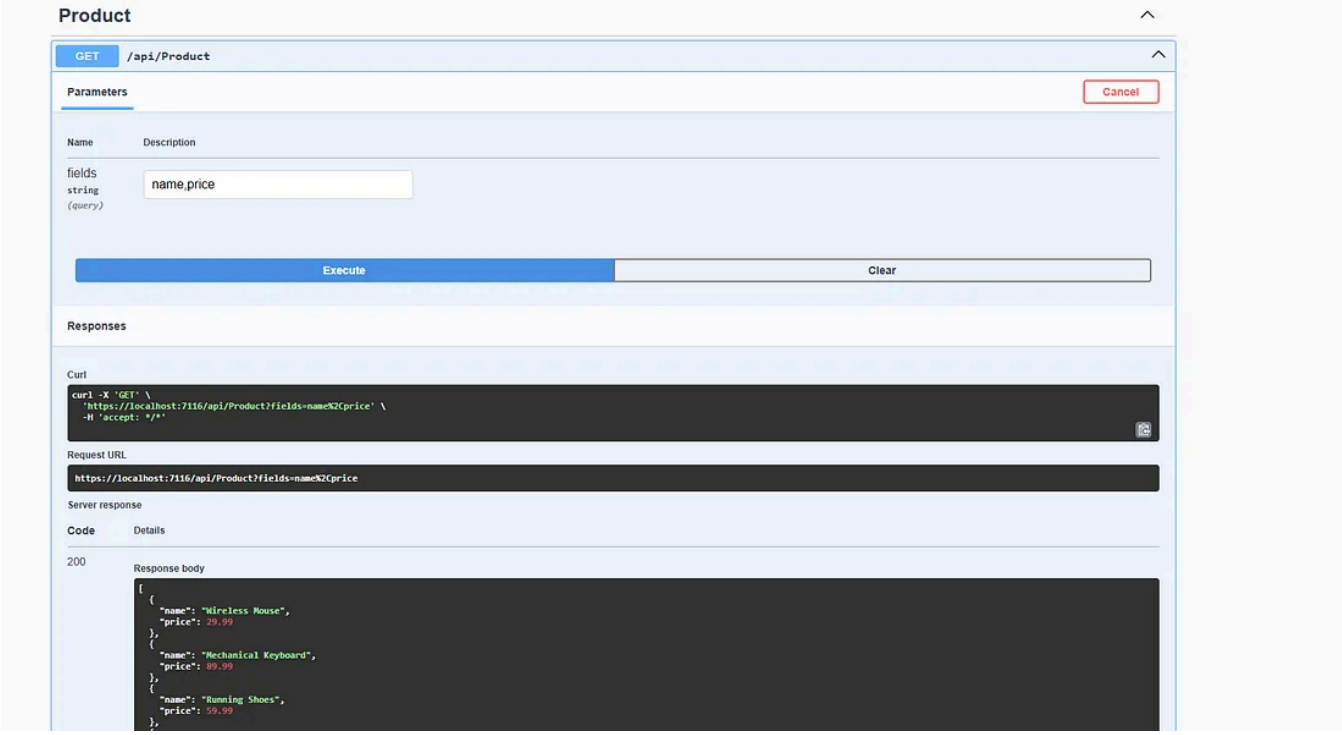
/api/books?fields=Title,Price

- `string? fields` : Parametrenin isteğe bağlı olduğunu belirtir (`nullable`)
- `fields ?? string.Empty` : Eğer `fields` null gelirse, boş string olarak davranılır.
- `ShapeData()` : Sadece istenen alanlara göre veri döndürür. Eğer `fields` boşsa, genellikle tüm alanlar döndürülür.

Şimdi swagger üzerinden API testimizi gerçekleştirelim.



Görüldüğü üzere field alanı boş ise tüm alanlar döndürülüyor. Şimdi sadece name ve price alanını dönmesini isteyelim.



Şimdi ise istenilen alanları dönmesini sağladık.

Konuyu bu şekilde özetleyebiliriz. Fakat daha iyi kavramak amacı ile görseldeki örneğin nasıl çalıştığını kodumuz üzerinden sırasıyla işleyelim.

```
[HttpGet]
public IActionResult GetBooks([FromQuery] string? fields)
{
    var products= ProductStore.Products;
    var shapedData = _dataShaper.ShapeData(products, fields ?? "");
    return Ok(shapedData);
}
```

ProductController üzerinden get isteği attığımızda yukarıdaki kod bloğu çalışacaktır. Query'den gelen kısma "?fields=name,price" alanı karşılık gelir. Bu da fields parametresine aktarılır.

- Mock dataları tutan ProductStore classındaki Products listesine "products" değişkeni atanır.
- _dataShaper örneği üzerinden ShapedData metoduna ulaşılır ve parametreleri verilir.

```
ShapeData(IEnumerable<T> entities, string fields)
```

entities → products

fields → name,price

```
var propertyInfos = typeof(T).GetProperties(BindingFlags.Public | BindingFlags
```

T türüne karşılık Product gelir.

- Product nesnesinin property'leri propertyInfos adlı değişkene atanır.

```
public int Id { get; set; }
public required string Name { get; set; }

public required string Brand { get; set; }
public required string Description { get; set; }
public decimal Price { get; set; }
public required string PictureUrl { get; set; }
public required string Type { get; set; }
public int QuantityInStock { get; set; }
```

```
var fieldList = string.IsNullOrWhiteSpace(fields)
    ? propertyInfos.Select(p => p.Name)
    : fields.Split(',', StringSplitOptions.RemoveEmptyEntries).Select(f => f.
```

fields parametresinin boş olup olmadığı kontrol edilir. Boş olmadığı için :

```
fields.Split(',', StringSplitOptions.RemoveEmptyEntries).Select(f => f.Trim());
```

satırı çalışır. fields (yani “name,price”) boşluklar atılıp bir dizi haline getirilir.

name,price → [“name”,”price”]

```
var shapedData = new List<ExpandoObject>();
```

Geriye dönecek olan shapedData için bir liste tanımlıyoruz.

```
foreach (var entity in entities)
{
    var shapedObject = new ExpandoObject() as IDictionary<string, object>;

    foreach (var field in fieldList)
    {
        var prop = propertyInfos.FirstOrDefault(p =>
            p.Name.Equals(field, StringComparison.InvariantCultureIgnoreCase))

        if (prop != null)
        {
            shapedObject[field] = prop.GetValue(entity);
        }
    }

    shapedData.Add((ExpandoObject)shapedObject);
}
```

- `fieldList` içinde belirtilen alanlar (örneğin `"name"` , `"price"`) üzerinde tek tek dönülür.
- Her `field` için, `propertyInfos` listesinden (`Product` entity'sine ait tüm property bilgileri) eşleşen bir özellik (property) bulunmaya çalışılır. Bu işlem, büyük/küçük harf duyarsız şekilde yapılır.
- Eğer eşleşen bir property (`prop`) bulunursa, o property'sinin `entity` içindeki değeri alınır ve `shapedObject` sözlüğüne eklenir.
- Burada sözlüğün anahtarı (`key`) alan adı (`field`), değeri ise o alanın `entity` içindeki değeridir.

```
shapedObject["price"] = 20;
```

- Bu, ilgili `Product` nesnesinin `Price` property'sinin değerinin `20` olduğu anlamına gelir.
- Bu işlem, `entities` listesindeki tüm nesneler için tekrarlanır. Her `entity` için oluşturulan `shapedObject`, `ExpandoObject` olarak `shapedData` listesine eklenir.
- Sonuç olarak, `shapedData` listesi, sadece istenilen alanları (örneğin `Name`, `Price`) içeren dinamik nesneleri barındırır ve bu liste geriye döndürülür.

```
return Ok(shapedData);
```

Controller kısmında shapedData döndürülerek istenilen amaca ulaşılır.

Data Shaping konusunu kendimce anlatmaya çalıştım. Umarım okuyan herkes için faydalı olur. Eleştiri ve yorumlarınızı belirtirseniz çok sevinirim. Herkese Kolay Gelsin!

Süleyman Meral

Software Engineering Student/ .NET Developer

Yazılım | Yazılım Mühendisliği | Aspnetcore

Edit profile

Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)