

Ocelot ile API Gateway Kullanımı: Microservice Mimarisinde Trafik Kontrolü

7 min read · 4 days ago



Süleyman Meral



Share



More

Herkese Merhaba! Mikroservis mimarisine geçiş yapan her geliştirici, eninde sonunda şu soruyla karşılaşır:

“Tüm bu servisleri dış dünyaya nasıl açacağım?”

İşte bu noktada devreye API Gateway kavramı girer. .NET dünyasında ise bu iş için Ocelot sıklıkla kullanılır.

Bu yazıda, Ocelot nedir, ne işe yarar ve nasıl kullanılır gibi sorulara yanıt verirken; aynı zamanda bir API Gateway’in mikroservis dünyasındaki önemini örneklerle ele alacağız.

Süleyman Meral



OCELOT ile API Gateway



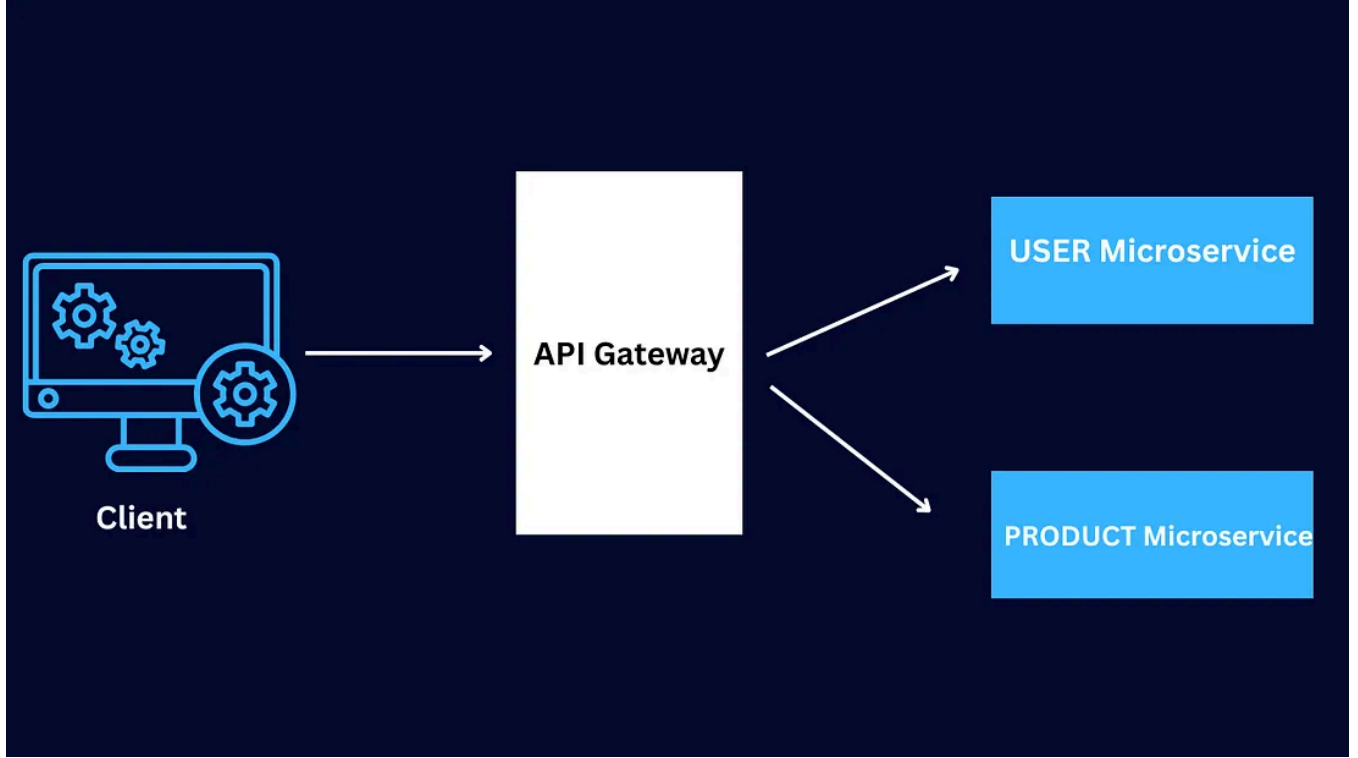
ASP.Net Core

mrslymn02@gmail.com

API Gateway Nedir?

API Gateway, mikroservislerin önünde duran bir “geçit”tir.

Client taraftan gelen istekler önce bu geçide gelir, ardından ilgili mikroservise yönlendirilir. (Örneğin React Uygulamamızdan client tarafa kullanıcı oluşturmak için bir request gönderdik. Bu request API Gateway’e gider daha sonra kullanıcı işlemleri ile ilgili User Microservice’ine yönlendirilir.)



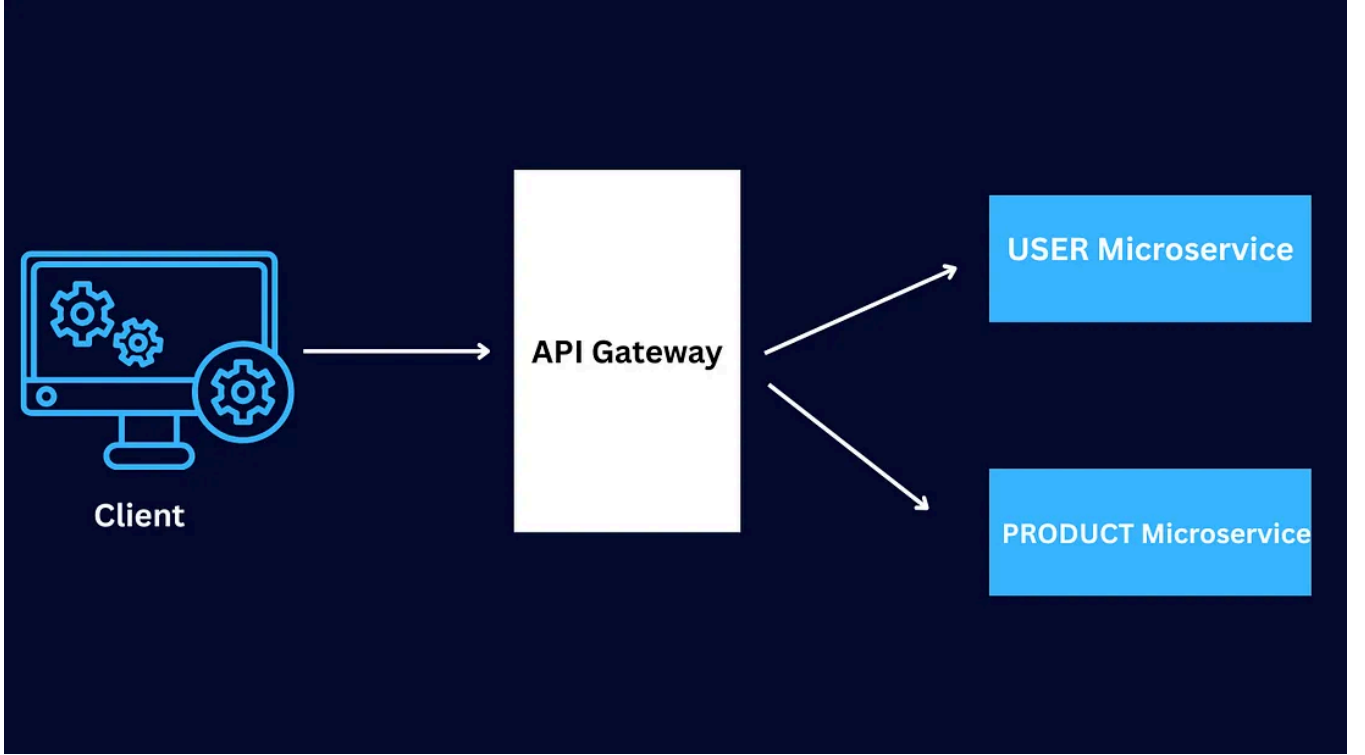
- İstekleri doğru mikroservise yönlendirir (Routing)
- Gelen JWT token’ı kontrol eder (authentication & authorization)
- Gerekirse isteği dönüştürür veya önbellekleme yapabilir.
- Karmaşık sistem yapısını client tarafa **tek bir API yüzü gibi** sunar

Örneğin “GET <http://apigateway.local/products>” client taraf API Gateway’e böyle bir istek gönderir. Gateway bu isteği şu şekilde dönüştürür :

“GET <http://productservice.local/api/v1/products>”

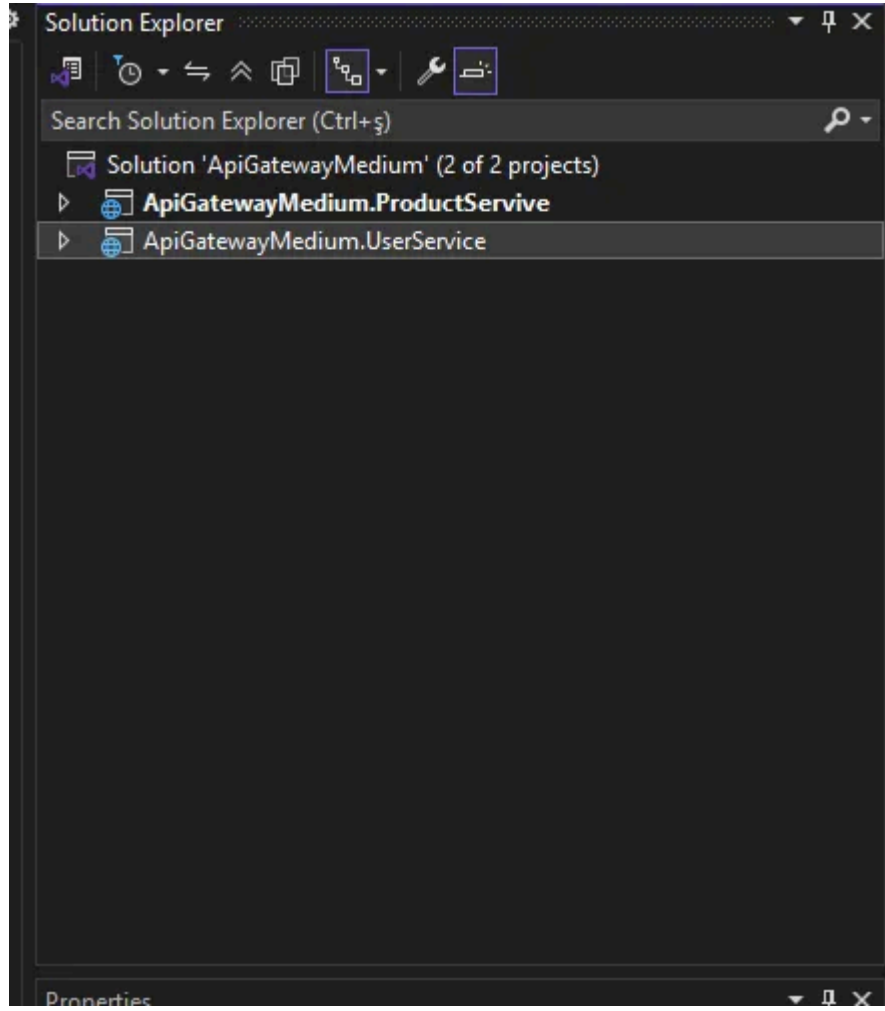
- Client, servislerin detayını bilmeden etkileşime geçmiş olur.
- Eğer bir servis adresini değiştirirseniz sadece Gateway ayarını değiştirmeniz yeterlidir. Bu sayede esneklik sağlar.

Teorik kısım üzerinde kısa bir anlatım gerçekleřtirdik. řimdi konuyu asıl pekiřtirmemizi saęlayacak bir proje örneęi yapalım. Senaryomuzda User ve Product adında 2 farklı Microservice'imiz olsun.

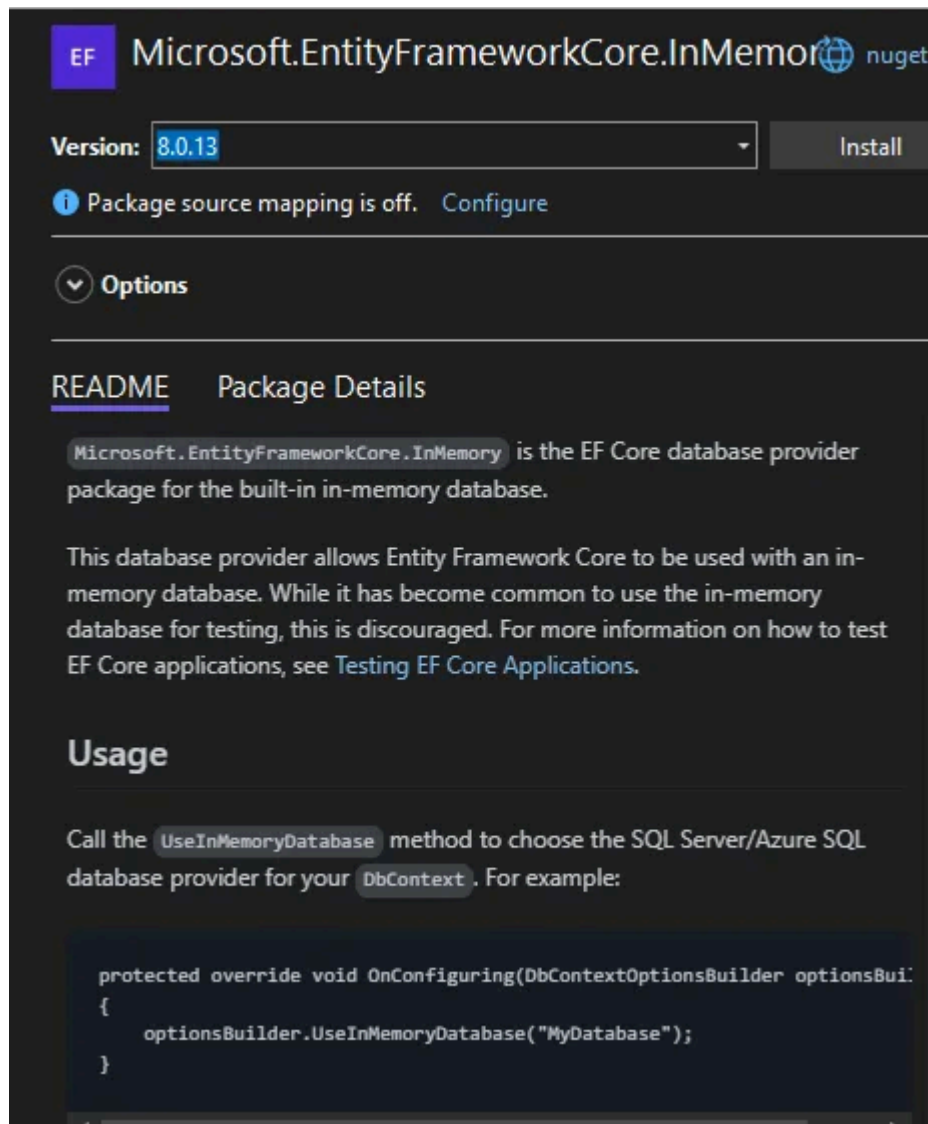


Client tarafta 3.Party uygulaması olan POSTMAN API kullanacaęız. İsteklerimizi oradan göndereceęiz.

Visual Studio üzerinden boş bir ASP.NET Core projesi oluřturalım.



Yukarıdaki gibi 2 farklı proje oluşturalım. Esnekliği daha iyi açıklamak maksadı ile Product servisimizde InMemory , User servisinde ise MSSQL Server kullanalım. Yaptığımız işlemleri detaylıca açıklamaya çalışacağım. Bu yazıyı okuyorsanız zaten belli bir seviyenin üstündesinizdir. Dilerseniz bu kısımları geçebilirsiniz.



Ben projemi .NET 8 ile oluşturdum. Long Term Support destekli versiyonları kullanmak daha sorunsuz olabilir. Fakat .NET 9 da kullansanız bir şey değişmeyecektir. Sürüm çakışması olmaması için projeyi oluşturduğunuz versiyon ile paketin aynı versiyon olduğuna dikkat edin. Yoksa hata alabilirsiniz.

InMemory ram üzerinden çalışır. Bu yüzden verileriniz veritabanına kayıt edilmez. Uygulama sonlanınca silinirler. Biz test amaçlı yaptığımız için bir sorun meydana getirmeyecektir.

Veritabanı için Context klasörü oluşturup, içerisine ApplicationDbContext classını oluşturalım.

```
using Microsoft.EntityFrameworkCore;

namespace ApiGatewayMedium.ProductService.Context;

public class ApplicationDbContext : DbContext
```

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

Projemiz küçük olacağı için minimal API yapısı kullanacağız. İşlemlerimizi program.cs dosyasında gerçekleştireceğiz. Program.cs 'e geçmeden önce Product modelimizi oluşturalım.

```
namespace ApiGatewayMedium.ProductService.Models;  
  
public class Product  
{  
    public int Id { get; set; }  
    public string? Title { get; set; }  
}
```


Şimdi bu modeli DbContext yapımıza DbSet olarak ekleyelim.

```
using ApiGatewayMedium.ProductService.Models;  
using Microsoft.EntityFrameworkCore;  
  
namespace ApiGatewayMedium.ProductService.Context;  
  
public class ApplicationDbContext : DbContext  
{  
    public ApplicationDbContext(DbContextOptions options) : base(options)  
    {  
    }  
  
    DbSet<Product> Products { get; set; }  
}
```

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>  
    options.UseInMemoryDatabase("MyDatabase"));
```

Program.cs dosyamıza DbContext'i ekleyelim.

```
app.MapGet("/", () => "Hello World!");
```

 (extension) `RouteHandlerBuilder` `IEndpointRouteBuilder`. `MapGet(string pattern, Delegate handler)` (+ 1 overload)
Adds a `RouteEndpoint` to the `IEndpointRouteBuilder` that matches HTTP GET requests for the specified pattern.

Returns:
A `RouteHandlerBuilder` that can be used to further customize the endpoint.

[GitHub Examples and Documentation \(Alt+O\)](#)

MapGet metoduna baktığımız zaman string tipte bir pattern (gelen HTTP isteğiyle eşleşecek olan url yoludur) ve bir Delegate(bu rotaya gelen isteği işleyecek olan metottur)(Genelde lambda ifadesi olur)istiyor.

```
app.MapGet("/products/getallproducts", (ApplicationDbContext context)=>
{
    var products = context.Products.ToList();
    return products;
});
```

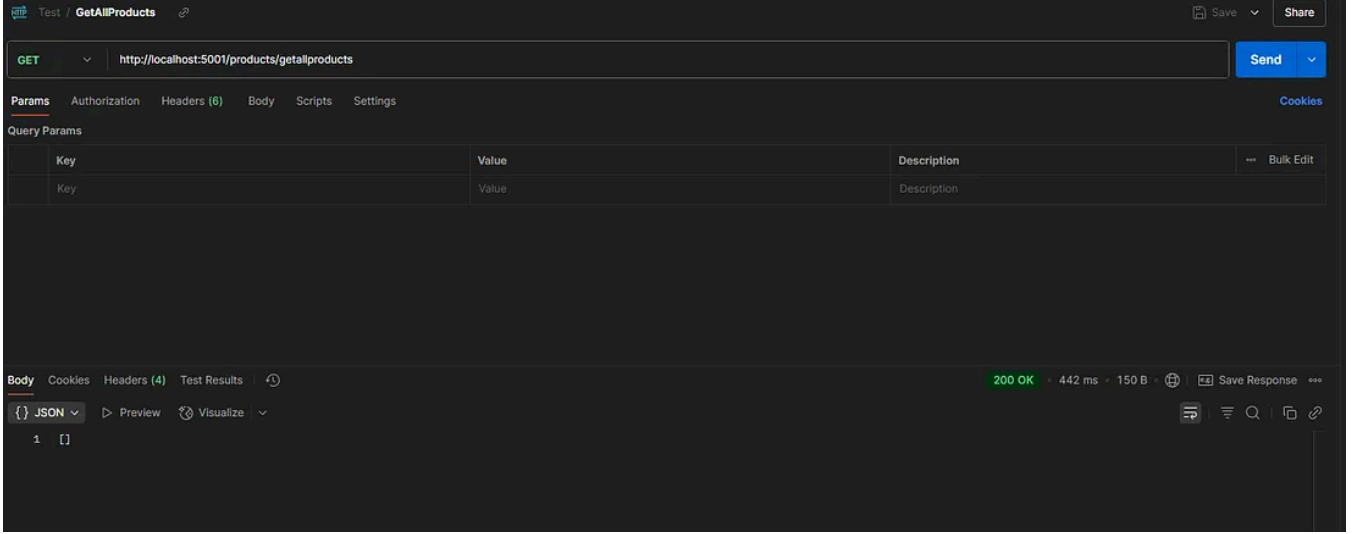
Fazla detaya girecek fakat şunu da eklemek istiyorum.

```
(ApplicationDbContext context)=>
{
    var products = context.Products.ToList();
    return products;
});
```

Bu ifade **lambda expression** şeklinde yazılmış bir `Func<ApplicationDbContext, IEnumerable<Product>>` delegate'idir.

```
(ApplicationDbContext context)
```

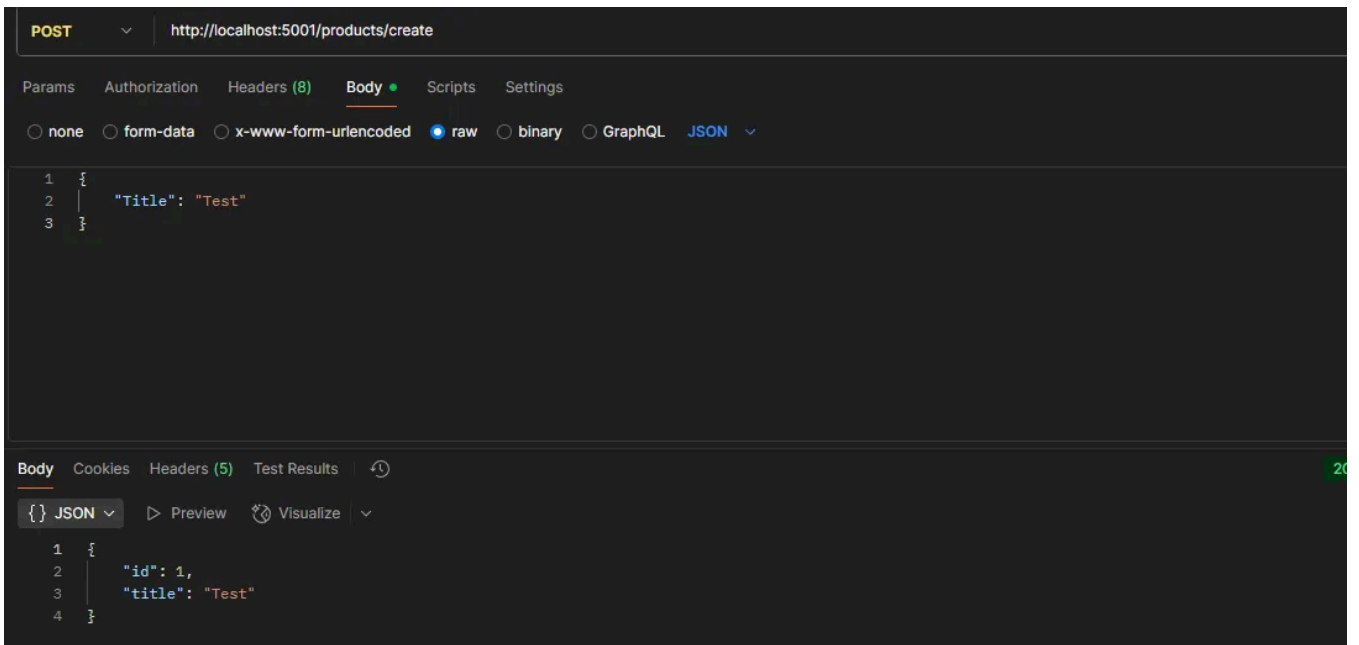
Metodun parametresi olarak düşünülebilir. Kalan ifadeyi de metodun body kısmı olarak düşünebiliriz. (İyi bir geliştirici olmanın püf noktalarından biri yazdığınız kodları en detayına kadar sorgulamanızdır).



Postman üzerinden request gönderiyoruz. Veri olmadığı için boş bir dizi dönüyor. Şimdi Create işlemini oluşturalım.

```
app.MapPost("/products/create", (Product product , ApplicationDbContext context)
{
    context.Products.Add(product);
    context.SaveChanges();
    return Results.Created($"{product.Id}", product);
});
```

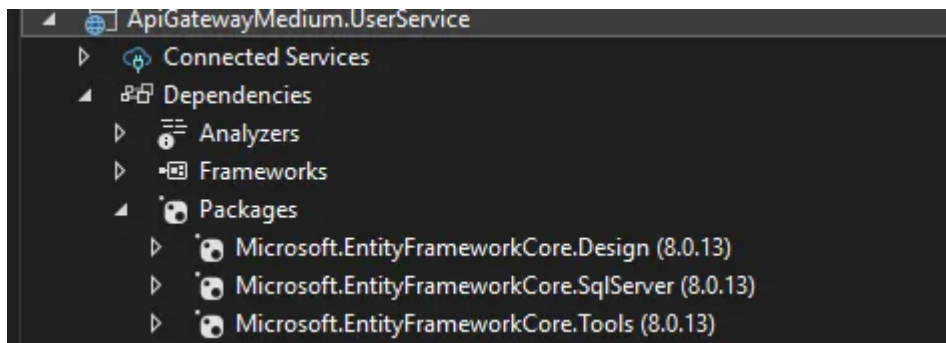
Varlıkları dış dünyaya direkt olarak açmak doğru bir yaklaşım değildir. DTO kullanılmalıdır. Biz memory düzeyinde basit bir işlem yaptığımız için böyle yaptık. User microservice'ini yazarken bu yapıyı kullanacağız. Varlık oluşturma işlemlerinde Created dönülebilir. 201 response kodu olarak döner. 201 sunucuda bir varlığın oluştuğunu ifade eder.



Tekrar GET isteği attığımızda verinin geldiğini görebilirsiniz. Fakat söylediğimiz gibi memory düzeyinde işlem yaptığımız için uygulama sonlanınca veriler kaybolacaktır.

Product servisi ile işlemimiz şu anlık bu kadar. Şimdi User servisine geçelim.

User microservice’imizde veritabanı olarak MSSQL kullanacağız. Dilerseniz farklı veri tabanları da kullanabilirsiniz.



Gerekli paketleri projemize ekleyelim.

Daha sonra context sınıfımızı yazalım ve User modelini oluşturalım. User modeli şifre ve kullanıcı içermeyecektir. Test amaçlı yaptığımız için sadece Name,Surname ve Title alanları olacak.

```
namespace ApiGatewayMedium.UserService.Models;

public class User
{
    public int Id { get; set; }
```

```
public string? Name { get; set; }  
public string? Title { get; set; }  
}
```

```
public class ApplicationDbContext : DbContext  
{  
    public ApplicationDbContext(DbContextOptions options) : base(options)  
    {  
    }  
  
    public DbSet<User> Users { get; set; }  
  
}
```

Program.cs üzerinden Context yapılandırmasını önceki işleme benzer şekilde yapalım.

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>  
    options.UseSqlServer(builder.Configuration.GetConnectionString("SqlServerCo
```

SqlServerConnectionString bizim bağlantı adresimizi tutacak. Bu adrese configuration üzerinden erişiyoruz. Bu yüzden appsettings.json dosyasına bu adresi belirtmemiz gerek.

```
appsettings.json  
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning"  
    }  
  },  
  "AllowedHosts": "*",  
  "ConnectionStrings": {  
    "SqlServerConnectionString": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=MicroServiceMediumDB;Integrated Security=True;Connect Timeout=30;Encrypt=False"  }  
}
```

Connection Keyi ile aynı şeyi yazdığımızdan emin olalım yoksa migration yaparken hata alırız.

Ayarlarımız bu kadar. Şimdi bir migration oluşturarak database'imizi ve tablomuzu oluşturalım.

```
Package Manager Console
Package source: All | Default project: ApiGatewayMedium.UserService
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party
additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.14.0.116

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration mgl
```

Package Manager üzerinden migration atıyoruz.(Startup Projesi hatası alabilirsiniz. User projesi üzerine sağ tıklayıp set as Startup proejct deyin. Hata düzelecektir)

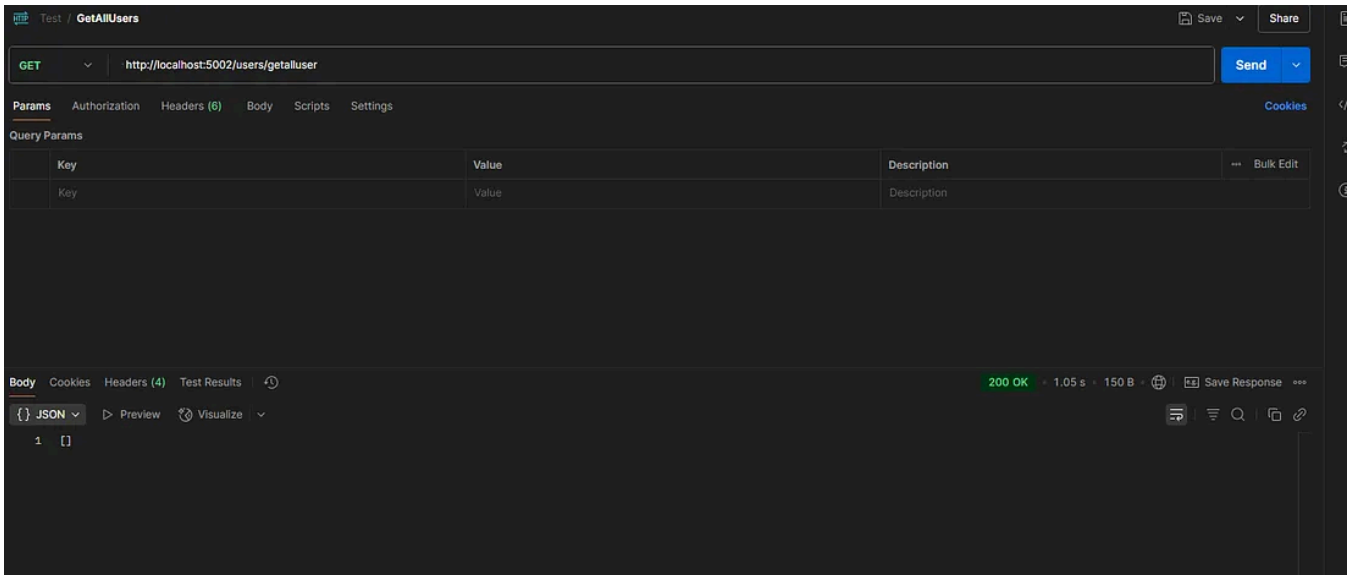
Migration eklendikten sonra “update-database” komutu ile değişiklikleri veri tabanında gözlemleyebiliriz.

Veritabanımız oluştu. Şimdi Endpointlerimizi yine program.cs üzerinden oluşturalım.

```
"profiles": {
  "http": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": false,
    "applicationUrl": "http://localhost:5002",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": false,
    "applicationUrl": "https://localhost:7143;http://localhost:5002",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
}
```

Portlarda problem yaşamamak için http adreslerinin portlarını aynı yapalım. Ayrıca projenizi ayağa kaldırdığınızda sürekli browser açılacaktır. launchBrowser ayarını false yapabilirsiniz.

```
app.MapGet("/users/getalluser", async (ApplicationDbContext context, Cancellation  
{  
    var users = await context.Users.ToListAsync();  
    return users;  
});
```



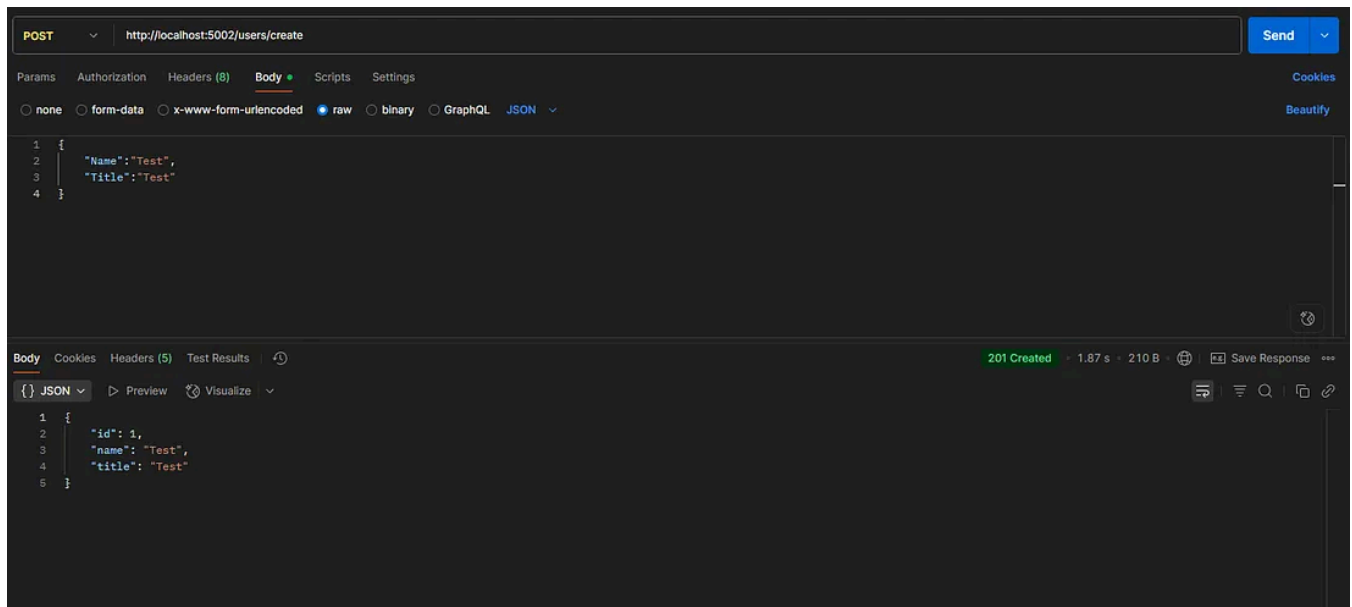
İlgili endpointe istek gönderdiğimizde 200 Response kodu döndüğünü görüyoruz. Bu işlemimizin başarılı olduğunu ifade ediyor. Fakat içerisinde veri olmadığı için şuan boş. Create işlemi için de bir endpoint oluşturalım. Bu işlem için bir DTO oluşturacağız.

```
namespace ApiGatewayMedium.UserService.DTOs  
{  
    public record CreateUserDto(string Name,  
                                string Title);  
}
```

Dto yapılarını record type olarak tanımlarsanız daha güvenli bir yapı oluşturursunuz. Immutable oldukları için bir kere set edildikten sonra değişmezler.

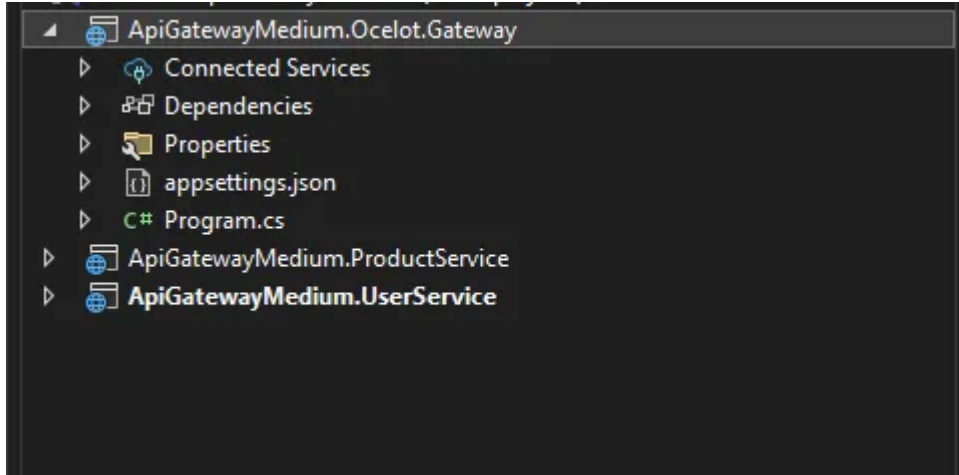
```
app.MapPost("/users/create", async(CreateUserDto createUserDto, ApplicationDbContext context) =>
{
    User user = new User
    {
        Name = createUserDto.Name,
        Title = createUserDto.Title
    };
    context.Users.Add(user);
    await context.SaveChangesAsync(cancellationToken);
    return Results.Created($" /users/{user.Id}", user);
});
```

CancellationToken istek atıldığında kullanıcının işlemi iptal etmesini sağlayan bir yapıdır. Add işlemi yapıldıktan sonra SaveChanges metodunu çağırmayı unutmayın. Add metodu verileri ram üzerinde tutar. SaveChanges metodu ile bunlar veritabanına yansır. Şimdi bu endpointimizi test edelim.



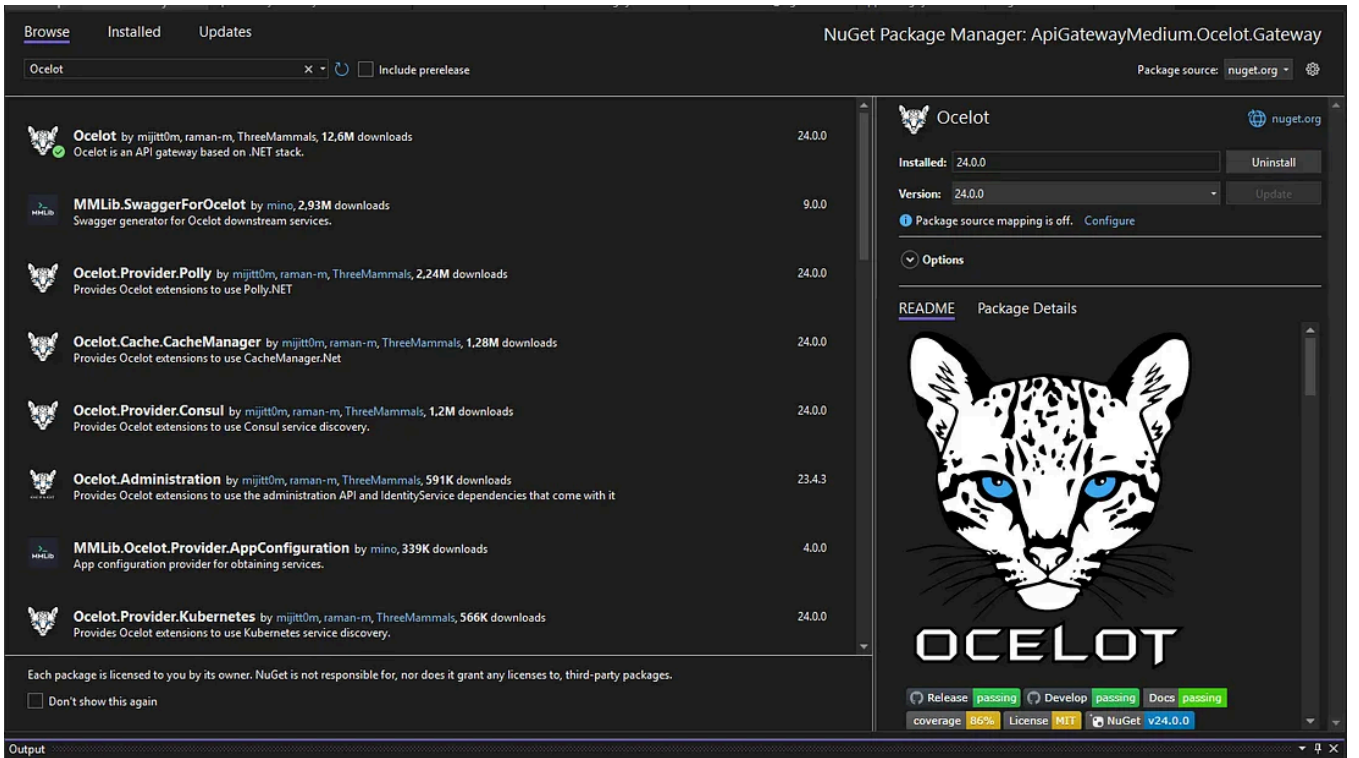
201 response code döndüğüne göre kaynağımız oluşturulmuştur. Get isteği yaparak bunu test edebilirsiniz.

Temel yapılar ile 2 Microservice'imizi oluşturduk. Şimdi asıl işimiz olan API Gateway'imizi oluşturalım.



Yapımız bu şekilde olacak. Gateway için Ocelot kullanacağız.

Bunun için Gateway projemize Ocelot NuGet paketini eklememiz gerekiyor.



Ocelot ayarlarını yapmak için ocelot.json adında bir dosya oluştuyorum.

If you want some example that actually does something use the following:

```
{
  "Routes": [
    {
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/ocelot/posts/{id}",
      "DownstreamPathTemplate": "/todos/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        { "Host": "jsonplaceholder.typicode.com", "Port": 443 }
      ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://api.mybusiness.com"
  }
}
```

Adımların detaylarına ocelot resmi sayfasındaki GetStarted tutorial sayfasını kullanarak erişebilirsiniz. Linki yazının sonuna ekleyeceğim.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/products/{everything}", // Gateway'den ilgili
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/api/products/{everything}", // Client taraftan
      "UpstreamHttpMethod": [ "GET", "POST", "DELETE" ]
    },
    {
      "DownstreamPathTemplate": "/users/{everything}", // Gateway'den ilgili mi
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5002
        }
      ],
      "UpstreamPathTemplate": "/api/users/{everything}", // Client taraftan Gat
      "UpstreamHttpMethod": [ "GET", "POST", "DELETE" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5000" // Gateway'in temel URL'si
  }
}
```

```
}  
}
```

ocelot.json dosyamızın yapılandırması bu olacak. İlgili istek önce client taraftan gatewaye gidecek, daha sonra ilgili microservice projesine yönlendirilecek.

Örneğin “GET <http://localhost:5000/api/products/42>” client taraftan Gateway’e gönderilen bu istek ocelot tarafından “GET <http://localhost:5001/products/42>” bu isteğe çevrilir. Buda ilgili mikroservisimize karşılık gelir.

Şimdi program.cs de ocelot ayarlarını gerçekleştirelim.

```
using Ocelot.DependencyInjection;  
using Ocelot.Middleware;  
  
var builder = WebApplication.CreateBuilder(args);  
builder.Configuration.AddJsonFile("ocelot.json");  
builder.Services.AddOcelot();  
  
var app = builder.Build();  
  
app.MapGet("/", () => "Hello World!");  
app.UseOcelot().Wait();  
  
app.Run();
```



```
C:\Users\suley\source\repos\... x + -
info: Microsoft.Hosting.Lifetime[14]
Now listening on: https://localhost:7100
info: Microsoft.Hosting.Lifetime[14]
Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\suley\source\repos\ApiGatewayMedium\ApiGatewayMedium.Ocelot.Gateway

C:\Users\suley\source\repos\... x + -
info: Microsoft.Hosting.Lifetime[14]
Now listening on: https://localhost:7012
info: Microsoft.Hosting.Lifetime[14]
Now listening on: http://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\suley\source\repos\ApiGatewayMedium\ApiGatewayMedium.ProductService

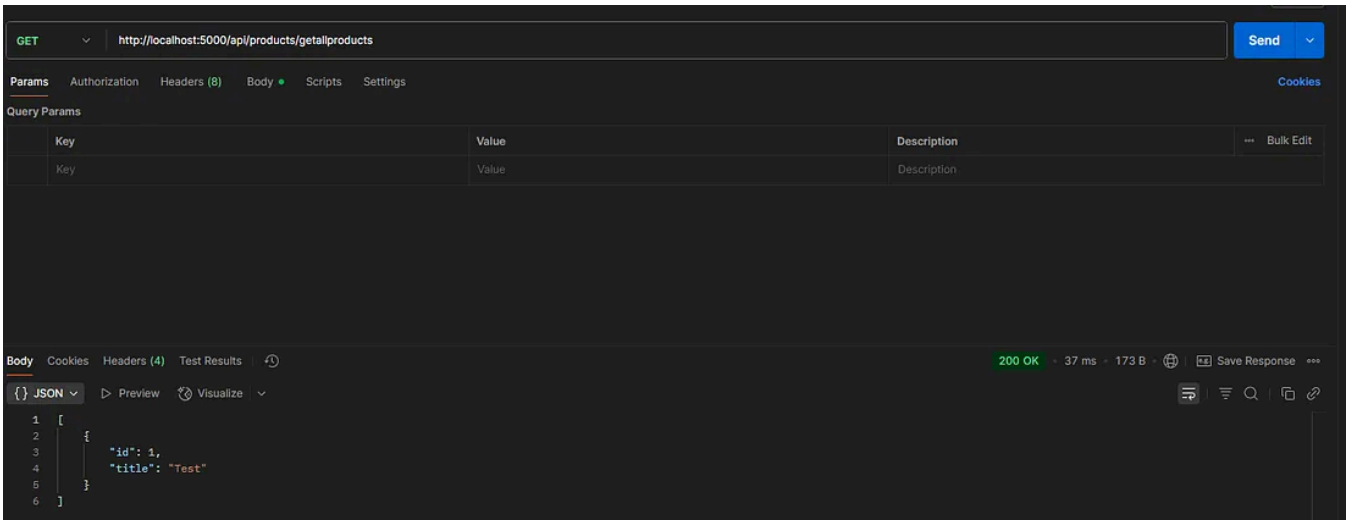
C:\Users\suley\source\repos\... x + -
info: Microsoft.Hosting.Lifetime[14]
Now listening on: https://localhost:7143
info: Microsoft.Hosting.Lifetime[14]
Now listening on: http://localhost:5002
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\suley\source\repos\ApiGatewayMedium\ApiGatewayMedium.UserService
```

3 projemizi de ayağa kaldıralım. Proje üstüne gelip ctrl+F5 yaparsanız ayrı ayrı projeleri ayağa kaldırabilirsiniz.

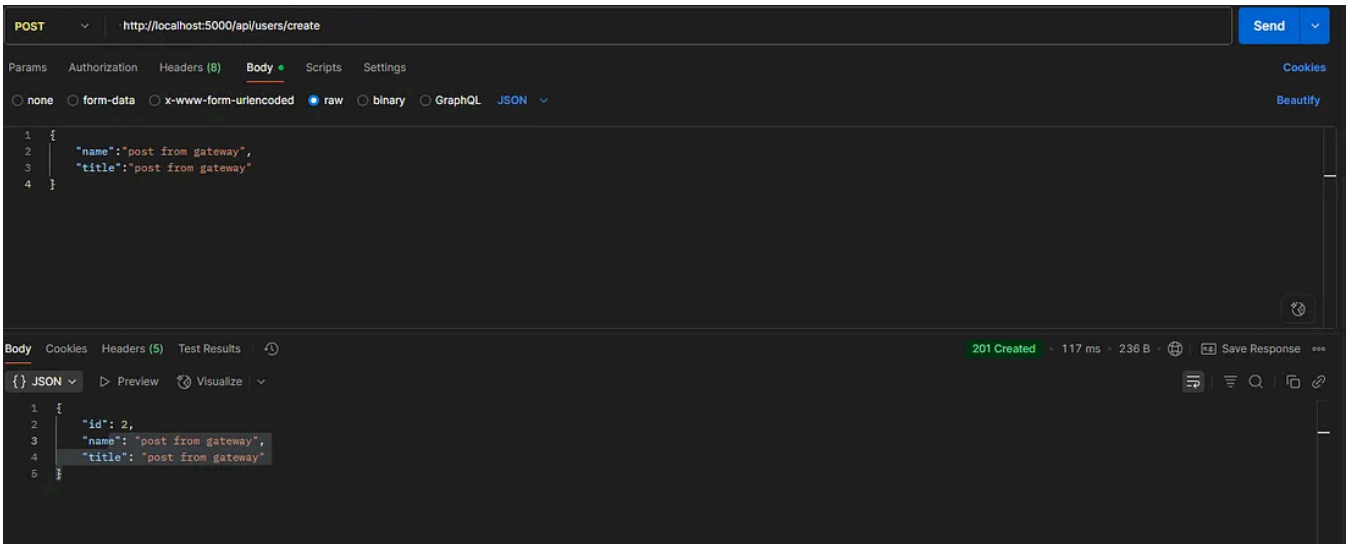
İşlemlerimiz tamam. Şimdi Postman üzerinden testimizi gerçekleştirelim. Gateway'in çalışıp çalışmadığını gözlemleyelim.

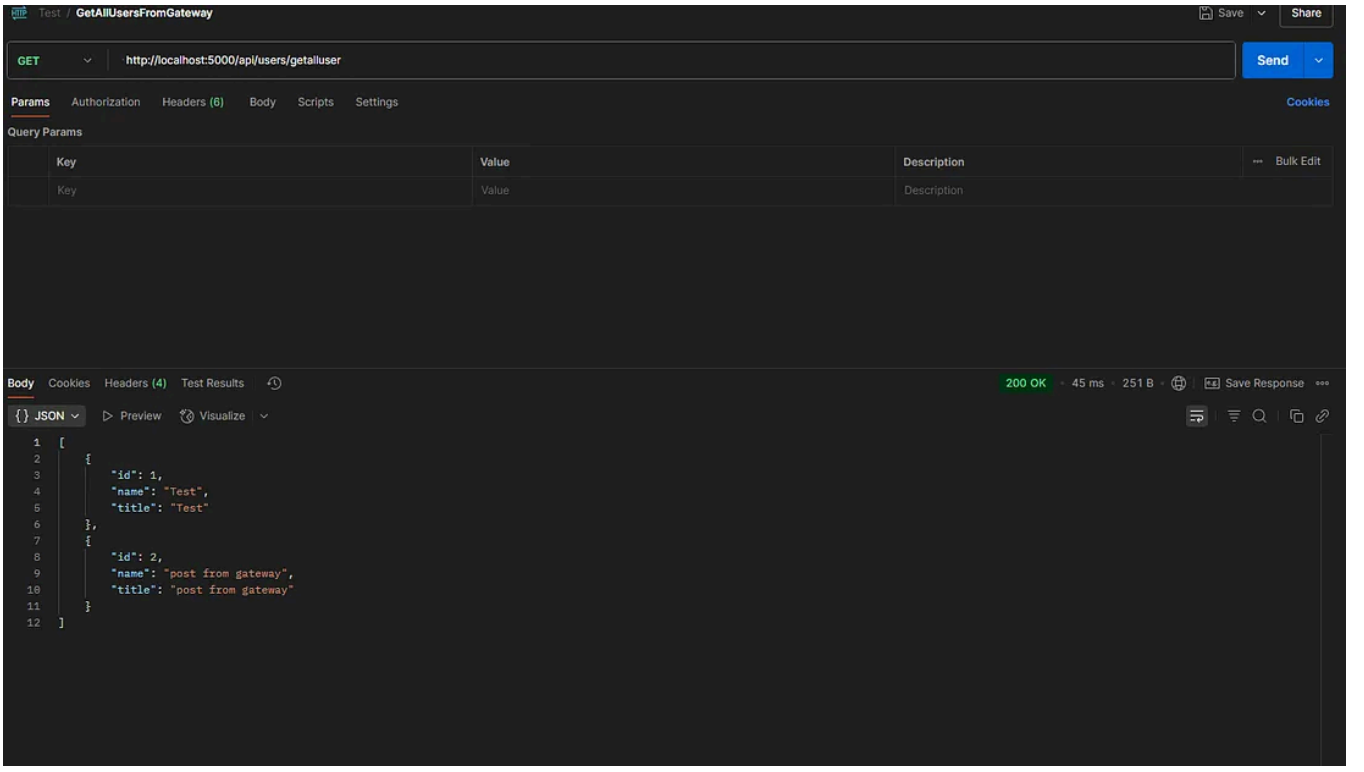
```
Test / CreateProducts
POST http://localhost:5001/products/create
Body
[{"Title": "Test"}]
201 Created - 348 ms - 199 B
{"id": 1, "title": "Test"}
```

Mikroservis projeme istek attığımda kaynak sorunsuz şekilde oluşuyor. Şimdi oluşan bu kaynağı gateway üzerinden getirelim.



Görüldüğü gibi başarılı bir şekilde geliyor. Birde gateway üzerinden user oluşturup gözlemleyelim.





Görsellerde gördüğümüz üzere Gateway'imiz sorunsuz şekilde çalışıyor. Burada karşılaşacağımız sorun tüm projelerin nasıl sürekli ayakta duracağı. Burada devreye Docker giriyor. Bu konu üzerinde de araştırma yapabilirsiniz.

Ocelot ile Gateway anlatımımız bu kadardı. Kendi cümlelerimle detaylandırıp anlatmaya çalıştım. Umarım faydalı olmuştur. Ocelot dökümantasyon linkini aşağıya bırakıyorum. Ek olarak http ile çalışırsanız daha iyi olur. Yoksa ssl hatası alabilirsiniz. Sorunuz olursa mail adresim üzerinden bana ulaşabilirsiniz. Herkese kolay gelsin!

Mail: mrlslymn02@gmail.com

Ocelot :<https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html>

Süleyman Meral

.NET Core Developer

Mikroservis Mimarisi

Net Core

Yazılım