

.NET Teknolojileri ile Mikroservis Mimarisi

TAMER YILDIRIM



Digital Vizyon
Akademi

İÇERİK

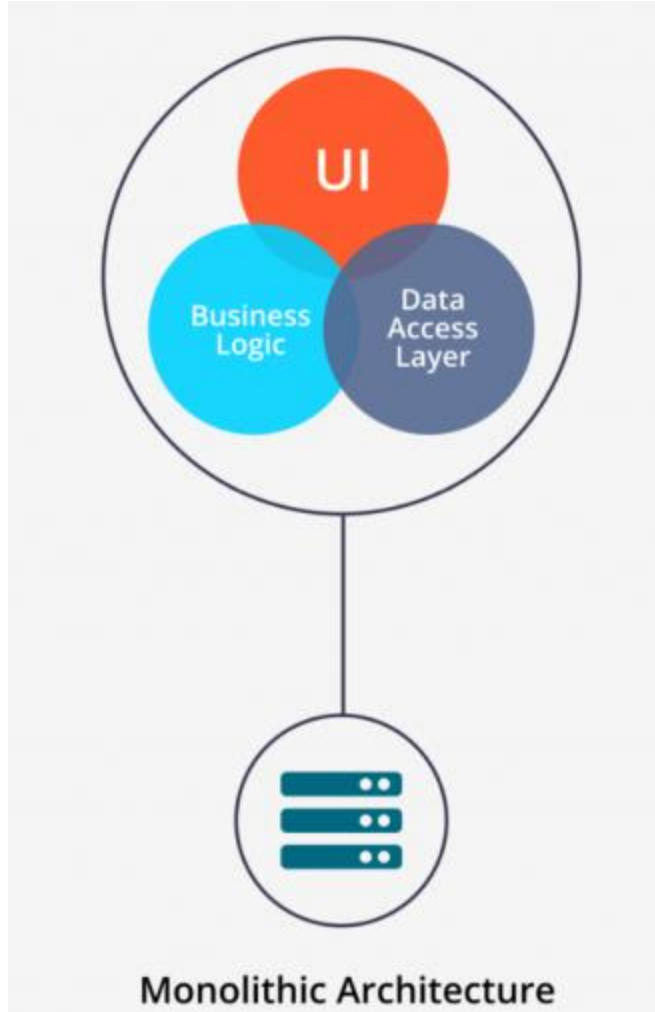
- 01 **Giriş**
Monolithic & Microservice
- 02 **API Gateway**
Ocelot- API Gateway
- 03 **Load Balancing**
Ocelot – Load Balancing
- 04 **Event Driven**
RabbitMQ – CAP

Monolithic Mimari Yaklaşım

Monolithic yaklaşım, bir sistemin/nesnenin/olgunun tek bir parça olacak şekilde tasarlanmasıdır. Monolithic mimari ise bu tasarımın stratejik yapılanmasıdır.

Monolithic yaklaşım, üretilecek sistemin/nesnenin/olgunun bileşenlerini(component) birbirlerine bağlı(interdependent) olarak ve kendi kendine yetecek(self-contained) şekilde tasarlanmasını sağlayan ve böylece tek bir bütünsel varlık olarak nihai sonuca varılmasını sağlayan yapılardır.

Monolithic Mimari Yaklaşım



Monolithic yaklaşımı benimsemiş uygulamalar tüm fonksiyonallikleri tek bir çatı altında geliştirilirler

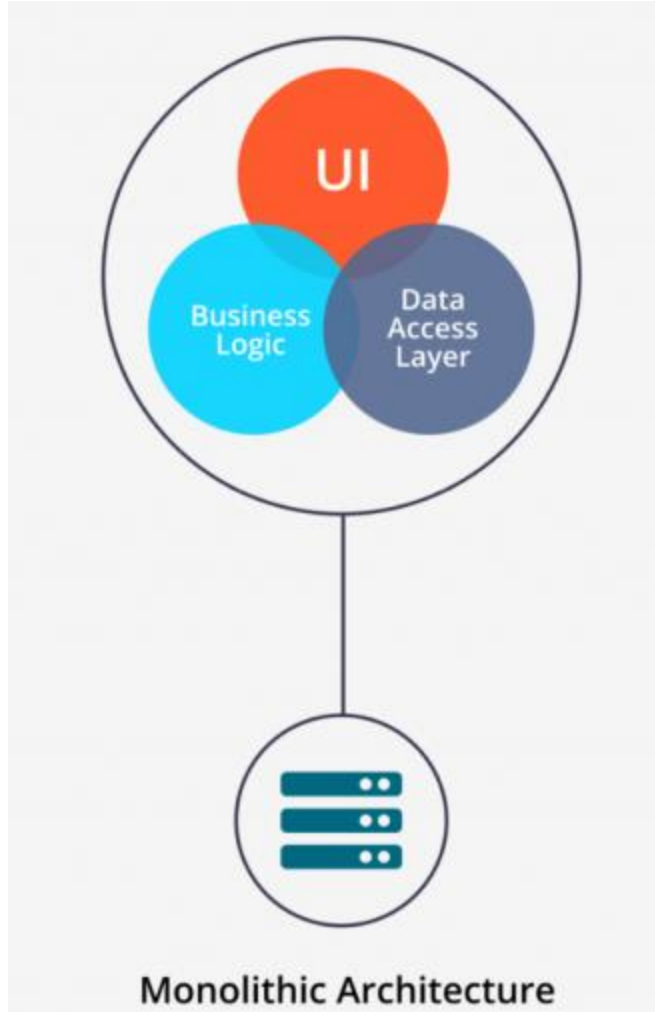
Avantajları ?

Dezavantajları ?



Digital Vizyon
Akademi

Monolithic Mimari Yaklaşım



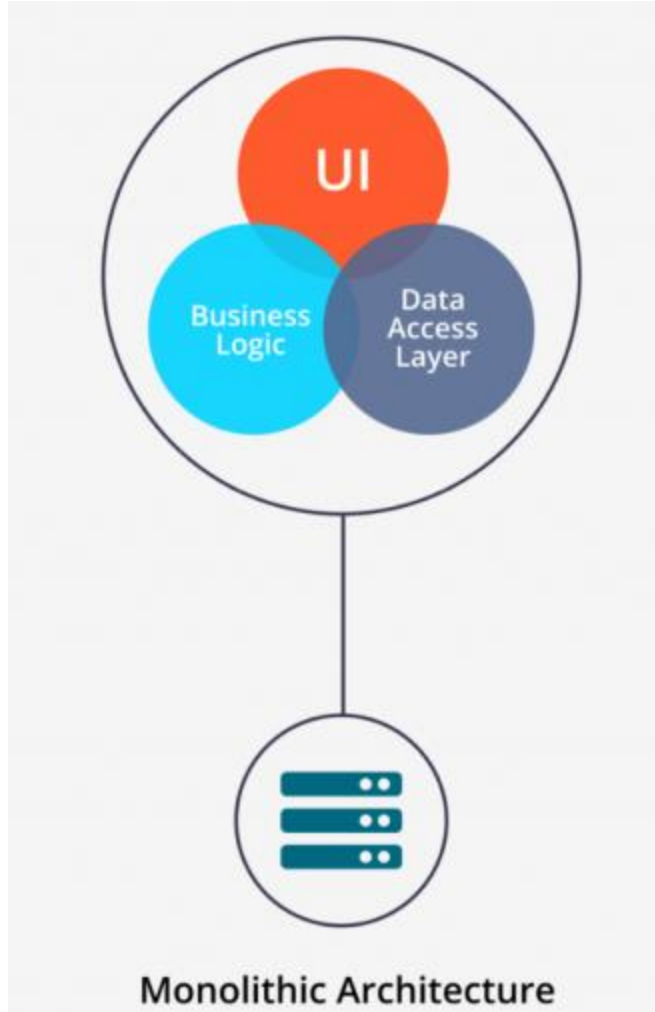
Avantajları

- Yönetilebilirliği, geliştirilebilirliği, bakımı ve monitoring'i(izleme) oldukça kolaydır.
- Küçük ve orta ölçekli projeler için geliştirilmesi hızlı ve maliyetsizdir.
- Component ve fonksiyonlar çalışma açısından kendi aralarında tutarlı ilişki kurabilmektedirler.
- Transaction yönetimi oldukça rahat ve kontrol edilebilirdir.



Digital Vizyon
Akademi

Monolithic Mimari Yaklaşım



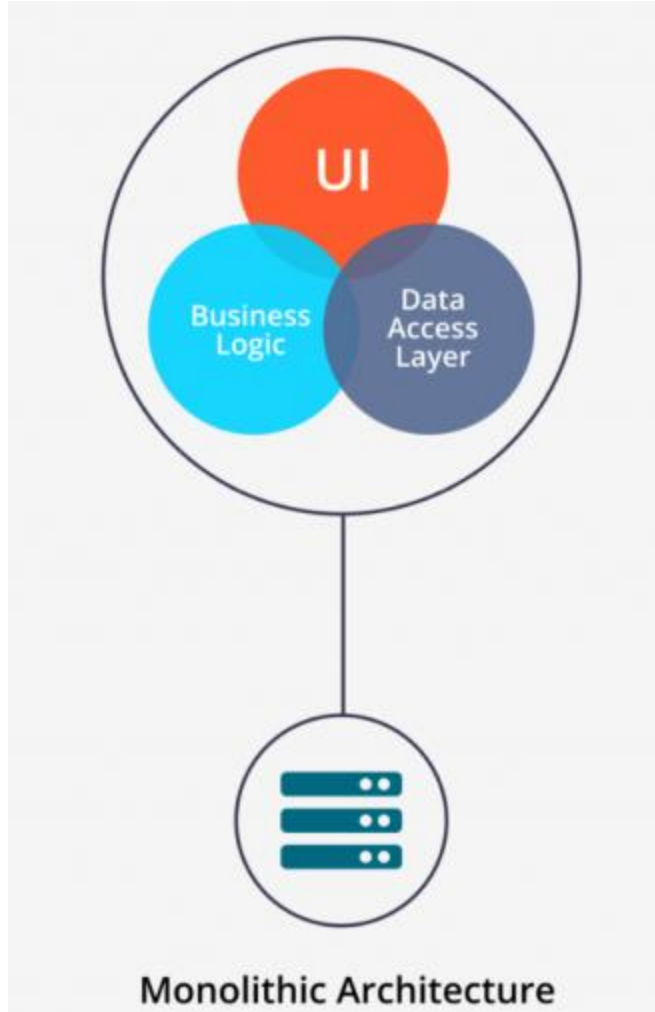
Dezavantajları

- Tüm hizmetler tek bir uygulama üzerinden sunulmaktadır. Böylece herhangi bir noktada düzeltme yahut geliştirme yapılması gerektiği takdirde uygulama baştan sona tekrar derlenmesi gerekmekte ve böylece uygulamanın varsa yayın durumu kısmi kesintilere gireceği anlamına gelmektedir.



Digital Vizyon
Akademi

Monolithic Mimari Yaklaşım



Dezavantajları

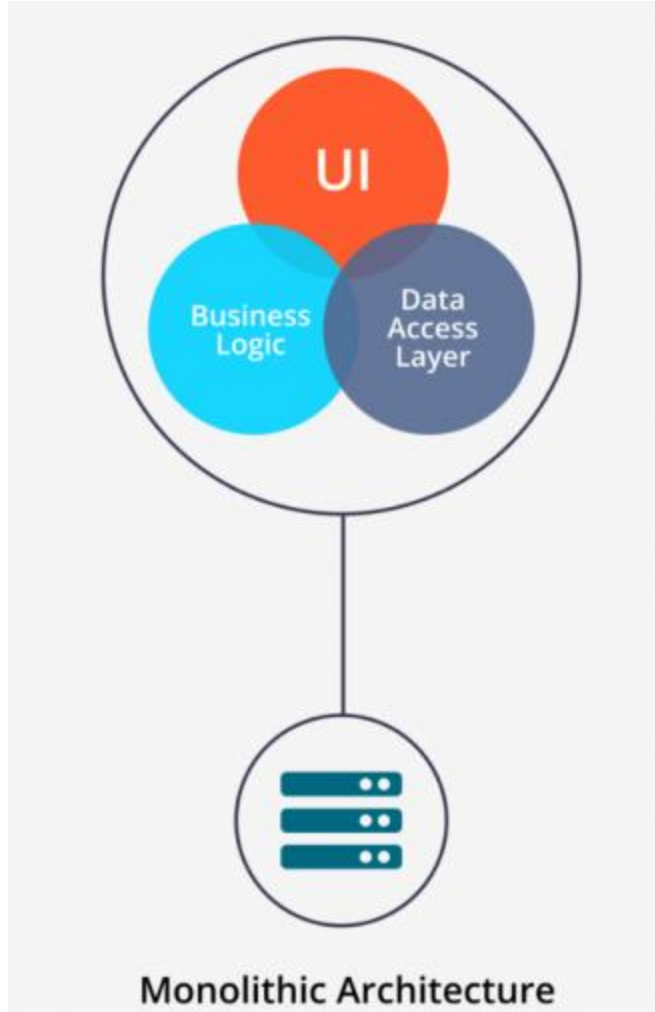
- Tüm bileşenler bütünsel bir parça içerisinde tek bir bütün olarak değerlendirilmektedir.

Bu durumda bir noktada yapılan çalışmanın alakasız başka bir noktayla olan teması yüzünden bloklanması ve süreçten etkilenmesi demektir.



Digital Vizyon
Akademi

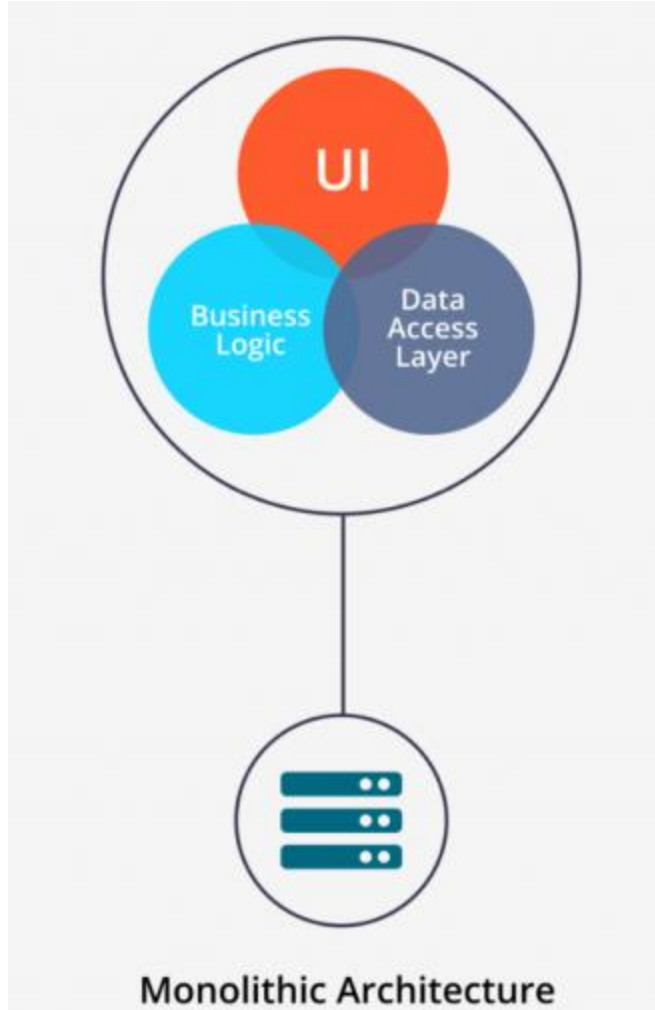
Monolithic Mimari Yaklaşım



Dezavantajları

- Monolithic yapılanmanın en kısır noktalarından biri bütünsel yaklaşımın getirdiği tek dil ve platform bağımlılığıdır. Uygulamanın bütünsel olarak inşa edilmesi tüm modüllerin aynı dil ve platformda inşa edilmesi mecburiyeti doğurmaktadır. Böylece farklı dil ve platformun kullanılamamasından dolayı ihtiyaç doğrultusunda dil ve platformun amacı dışına çıkılabilmekte ve bir çok angaryaya sebep olunabilmektedir

Monolithic Mimari Yaklaşım

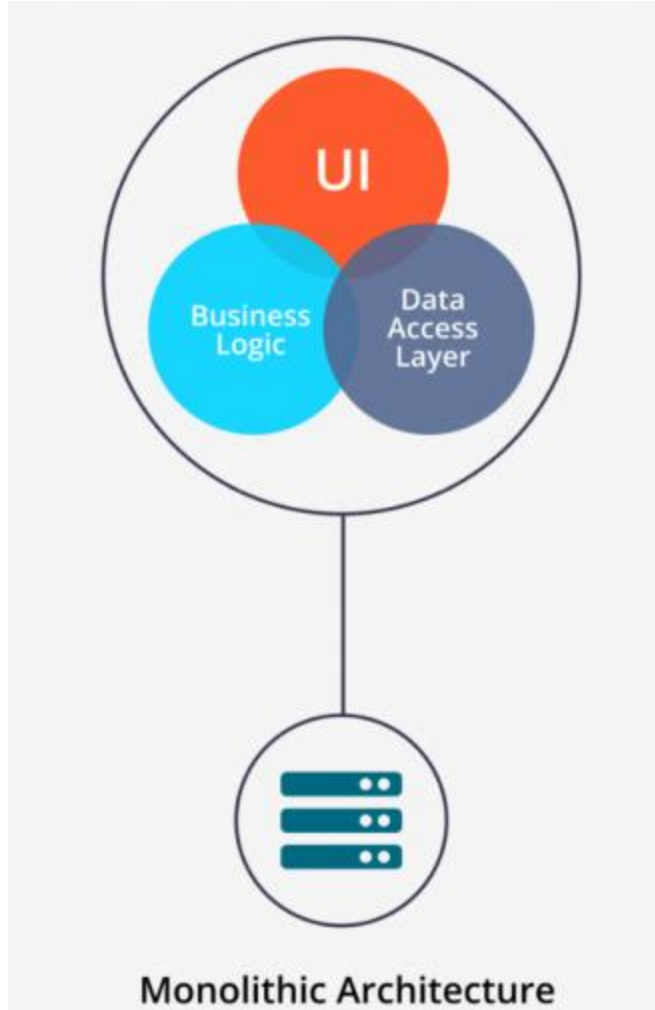


Dezavantajları

- Takım çalışmalarında birden fazla kişi tarafından geliştirilen uygulamalarda ister istemez birçok kod ve yapı karmaşası meydana gelmektedir.

Misal; ameliyat masasında mideden ameliyat olan bir hastaya biryandan da dış doktoru tarafından dolgu yapılması ne kadar sıkıntılı bir süreçse projelerde de benzer sıkıntılı süreçler ve gerginlikler yaşanabilmektedir.

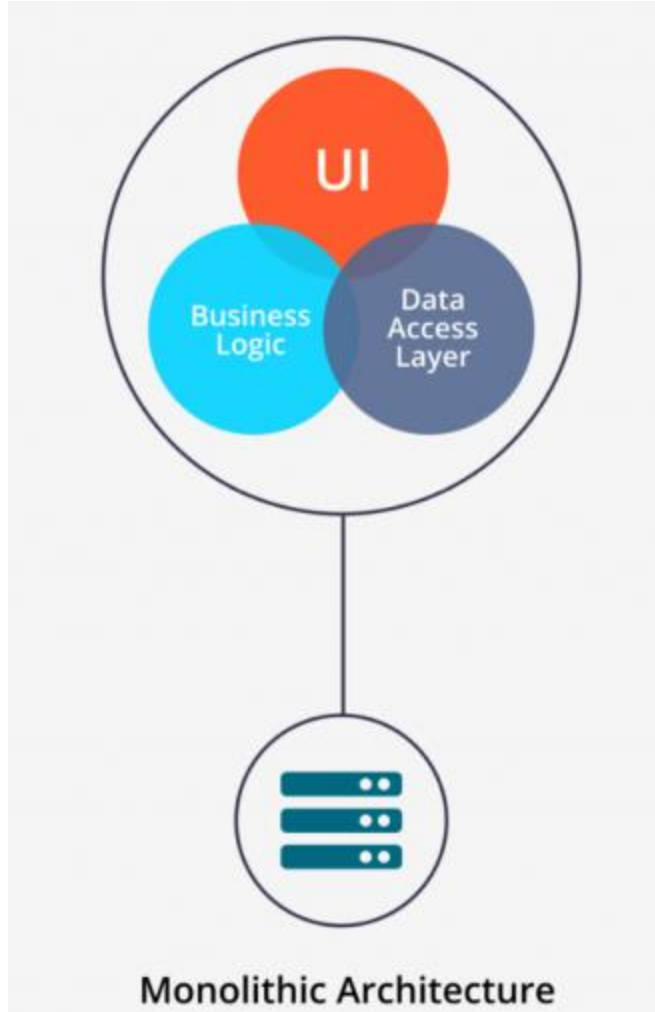
Monolithic Mimari Yaklaşım



Dezavantajları

- Uygulama tek çatı altında geliştirileceği için tüm component ve modüller kendi aralarında sıkı bağıllık göstereceklerdir. Böylece yukarıdaki misalde olduğu gibi diş doktoru dişe dolgu yaparken midede rahatsızlığın çıkma ihtimali gibi durumlar meydana gelebilecektir. Nasıl diş ile mide componentleri gevşek bağıllıkla bir arada çalışabiliyorsa, bizlerde yazılımları aynı modelde geliştirmeye özen göstermeliyiz.

Monolithic Mimari Yaklaşım



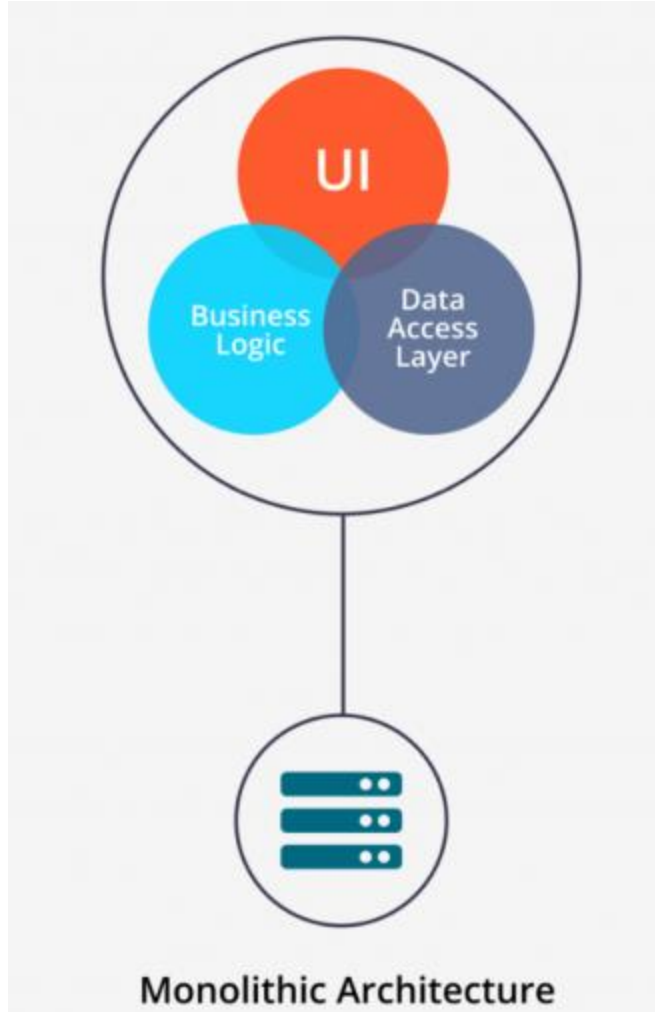
Dezavantajları

- Bu bağımlılıklardan kaynaklı olarak ufak bir noktadaki değişiklik başka alanlarda yeni değişikliklere sebep olabilmektedir.
- Versiyon yönetimi zorlaşır.



Digital Vizyon
Akademi

Monolithic Mimari Yaklaşım



Görüldüğü üzere monolithic mimarisi götürüsü getirisinden fazla olmakla birlikte artık yeni nesil bir yaklaşıma yerini bırakma noktasına gelmiş bulunmaktadır.

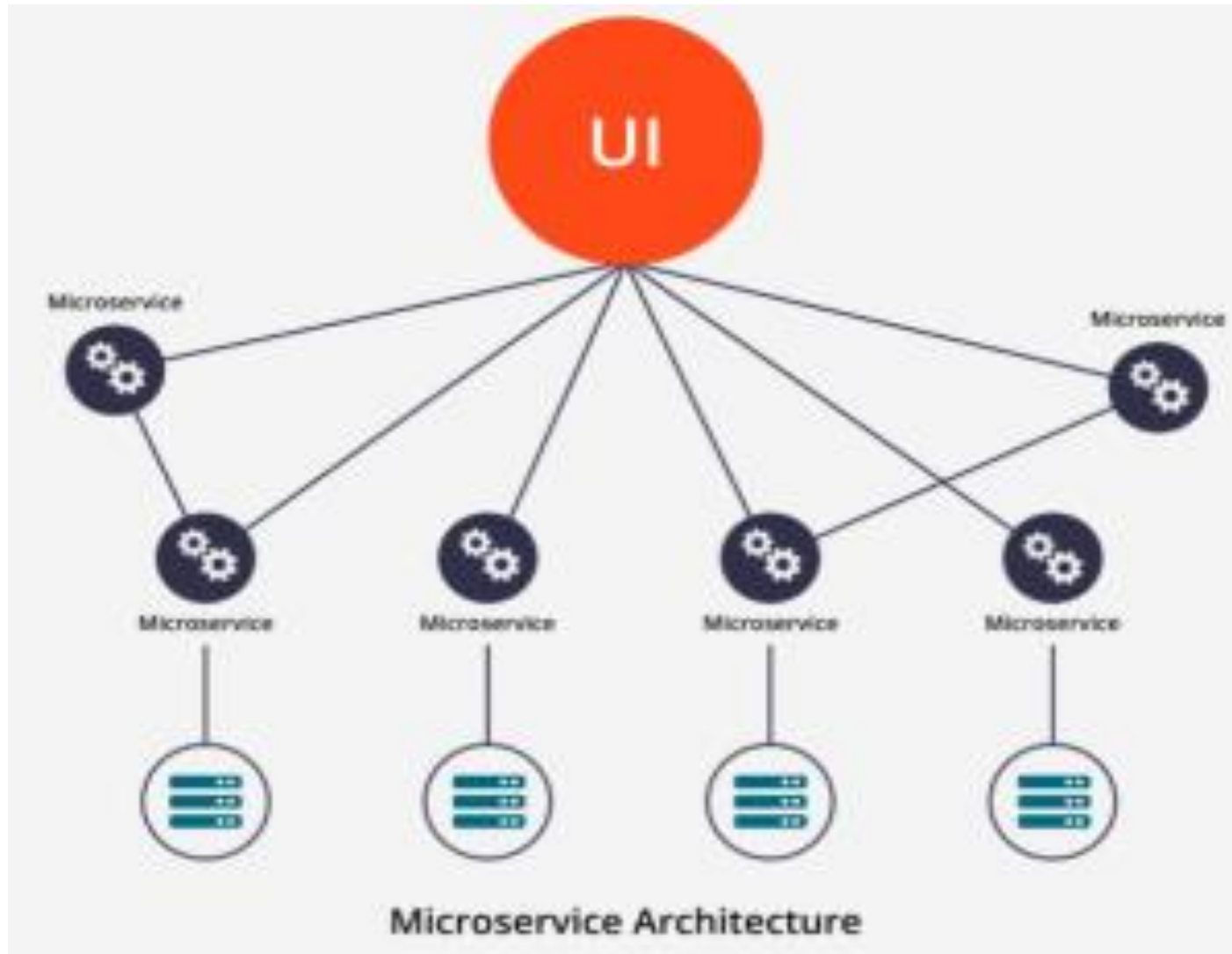


Microservice Mimarisi 😊

Microservice mimarilerine odaklanabilmek için öncelikle monolithic yaklaşımın temel bir prensibi çiğnediğinin farkında olunması gerekmektedir. Bu prensip sürdürülebilirlik ilkesidir.

Sürdürülebilirlik; bir yazılımın, üzerinde yapılan tüm değişiklik yahut onarma faaliyetleri esnasında bile, verdiği hizmetin bütününde bir aksaklık olmaması ve sistemin kesintiye uğramaksızın her an çalışabilir vaziyette olması demektir.

Microservice Mimarisi 😊



Microservice Mimarisi 😊

Microservice mimari, birbirinden bağımsız olarak çalışan ve birbirleriyle haberleşerek bir bütün olarak hareket eden servis(ler) yapılanmasıdır. Her servisin bir diğerinden bağımsız olarak iş mantığını yürütmesi ve bir başka servis ile ilgilenmemesi, bir onarım yahut restorasyon durumunda uygulamanın bütününe etkilemeyeceğinden dolayı sürdürülebilirlik ilkesi desteklenmiş olacak, böylece monolithic yaklaşımın yarattığı karmaşıklık ortadan kaldırılmış ve yönetimi daha da kolaylaştırılmış olacaktır.

Microservice Mimarisi 😊

Burada esas olan, her bir servisin bir diğerinden bağımsız olmasının geliştirme ve düzenleme operasyonlarında getirisiidir.

Örneğin; Bir e-ticaret uygulamasında ürün işlemlerinin, sepetin ve ödeme sisteminin ayrı servisler tarafından gerçekleştirildiğini düşünürsek, süreçte ödeme sistemindeki oluşan herhangi bir aksaklık yahut restorasyon sistemin bütününe değil sadece o servisi etkileyeceğinden dolayı haliyle sadece o servisle ilgilenilmesi yeterli olacaktır.

Microservice Mimarisi 😊

Bu durumda sistem bütünsel olarak işlevselliğe devam edecek lakin kesintiye sadece ilgili servis uğramış olacaktır. Haliyle bizler sistemin kendisinden ziyade local olarak sadece tek bir servisi ile ilgilenerek gerekli onarımı sağlayabilir, hızlıca testlere tabii tutabilir ve monolithic yapılanmalarda olduğu gibi uygulamayı topyekün derleme ve yayınlama ihtiyacını duymaksızın kısa zamanda yeni sürümle hizmete devam edebiliriz.

Microservice Mimarisi 😊

“Microservice mimarisi, uygulamayı bir bütün olarak geliştirmek yerine, küçük parçalar halinde geliştirilmesini amaçlayan bir felsefedir!”

Microservice Mimarisi 😊

Microservice mimari yaklaşımı uygulamayı dil ve platformdan bağımsız bir şekilde farklı veri depolama birimleri ve teknolojileri kullanılabilecek şekilde geliştirmemize imkan sağlayacak esneklik sunmaktadır.

Microservice Mimarisi 😊

Ayrıca her bir hizmetin küçük ve bağımsız servisler olarak tasarlanması aynı zamanda denetim ve uyum süreçlerini basit prensiplere dayandıran Agile disiplinininde uygulanmasını kolaylaştırmakta ve bu disiplin çerçevesinde geliştirilen uygulama küçük ve farklı alanlarda sorumluluklarını paylaşan ekiplerce inşa edilebilmektedir.

Microservice Mimarisi 😊

Monolithic uygulamalar yapısı itibariyle dikey genişletilmeye uygundur. Dolayısıyla bu durum donanımsal sınırlılıklar getirmektedir. Lakin microservice yapılanması ihtiyaca dönük dikey ve yatay genişletilebilmekte ve böylece daha net ölçeklendirilebilmektedir. Hatta her bir service ayrı ayrı ölçeklendirilebileceği için yaygın olarak bulut(cloud) ve sanallaştırma teknolojileri kullanılmaktadır.

Microservice Mimarisi 😊

Görüldüğü üzere microservice yapılanması birçok olumlu yönden hayatımıza katkıda bulunmaktadır.

Şimdi bu mimari yapının **avantaj** ve **dezavantajlarına** göz atalım;



Microservice Mimarisi 😊

Avantajları

- Uygulama boyutundan bağımsız olmak üzere yeni bir özelliğin eklenmesi yahut mevcutiyeğin bakımı sadece ilgili servislerle ilgilendirme gerektireceğinden dolayı oldukça kolaydır.
- Ekip çalışmasına yatkındır. Özellikle ekibe yeni katılım gösteren arkadaşların devasa bir proje ve kod içerisinde kaybolmasının önüne geçmekte, sadece ilgileneceği servisin kaynağını çözümülemesi gerekmektedir.

Microservice Mimarisi 😊

Avantajları

- Uygulama yapılan işlemler neticesinde servislerin birbirlerinden bağımsız olması tek başına scale edilebilmesini sağlamaktadır.
- Versiyon yönetimi oldukça kolaydır.
- Her bir service ihtiyaca binaen farklı dil ve platformda yazılabilmektedir.

Microservice Mimarisi 😊

Dezavantajları

- Birden fazla service ve birden fazla veritabanı söz konusu olacağı için transaction yönetimi zorlaşacaktır.
- Servislerin yönetilebilirliği ve monitoringi zorlaşacaktır.

Microservice Mimarisi 😊

Bu durumda her yeni başlayan projenin microservice temelli Atılması pek de doğru olmayacaktır. Genel olarak **bir proje temellendirilmesinin varsayılarda monolithic olarak tasarlanması** ve projenin büyüklüğü doğrultusunda **ihtiyaca binaen microservice yapılanmasına geçiş** tavsiye edilmektedir.

API Gateway

API Gateway, microservice yaklaşımını benimseyen bir uygulamada, client tarafından gelen istekleri ilgili servislere yönlendirme sorumluluğunu üstlenir.



API Gateway

API Gateway ile neler yapılabilir?

Authentication ve Authorization

İşlevsel sorumluluğu parça parça üstlenen servislere erişim api gateway üzerinden dolaylı yolla olacağından dolayı, kimlik ve yetki doğrulama operasyonları sadece api gateway'de yapılandırılabilir.

Logging

Servislere yapılan istekler hakkında detaylı loglamalar gerçekleştirilebilir ve böylece hangi servis, kim tarafından, ne kadar yoğunlukta işlevsellik gösteriyor vs. gibi istatistiksel bilgiler edinilebilir.

API Gateway

API Gateway ile neler yapılabilir?

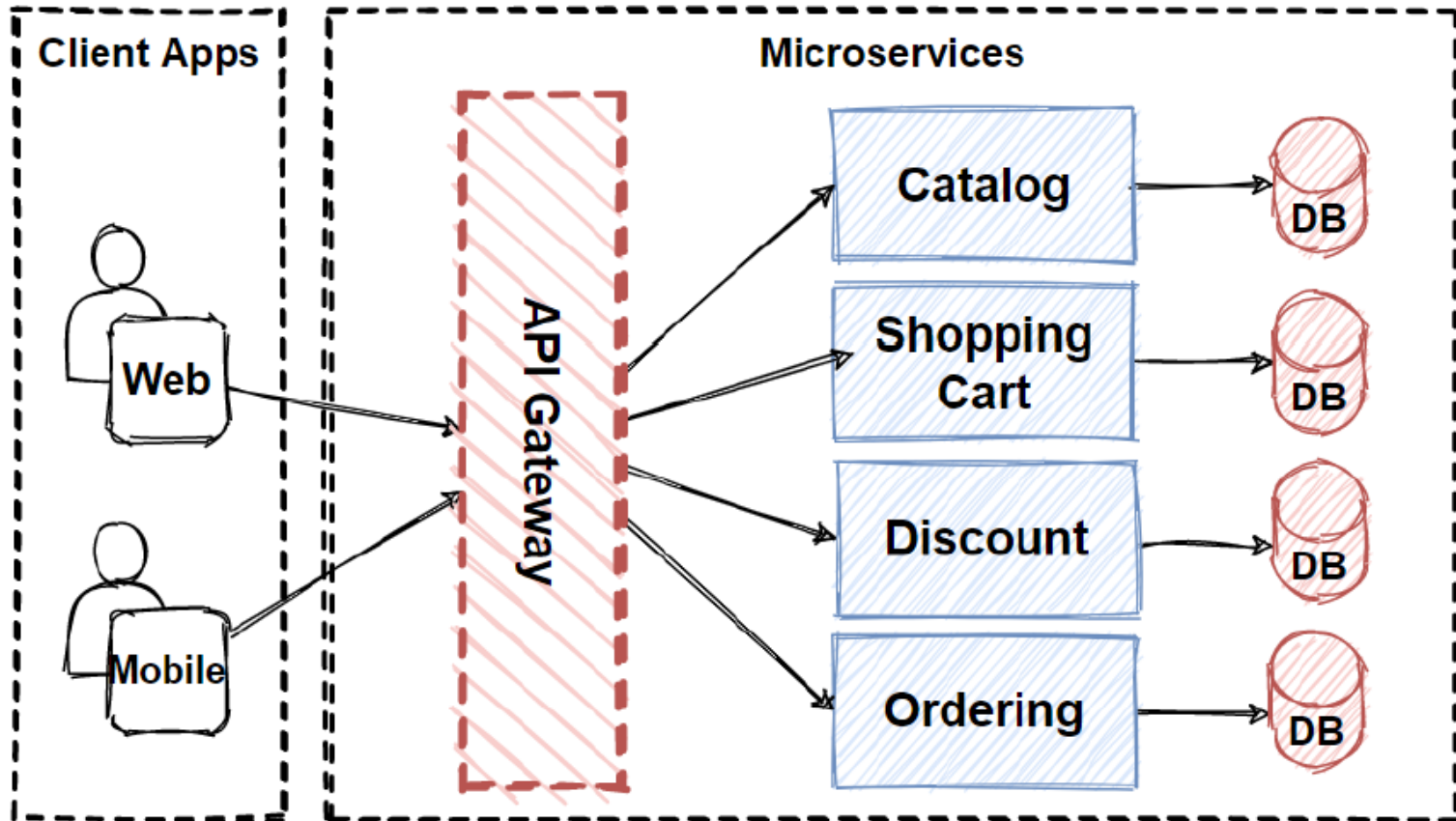
Response Caching

API gateway üzerinden, servislere gelen talepler neticesinde üretilen çıktıları cacheleyebilir ve böylece servis maliyetleri düşürülebilir.

Routing

Servislerin adresleri farklı şekilde kapsüllenebilir ve bu kapsül üzerinden clientlar ilgili esas routelara yönlendirilebilir.

API Gateway



Peki API Gateway'in **Avantaj** ve **Dezavantajları** Nelerdir?

API Gateway

Avantajları

- Client'ın ihtiyacı doğrultusunda birden fazla servis tarafından üretilecek olan datayı tek bir request – response ile üretilmesini sağlayarak daha az maliyetli bir kullanıcı deneyimi ortaya koyar.
- Authentication, authorization, logging, security, routing vs. gibi cross cutting concern kavramlarının tek elden yönetilmesini sağlar.

API Gateway

Avantajları

- En önemlisi clientları, uygulamanın microservislere nasıl bölündüğü hususunda düşünmekten izole eder.

API Gateway

Dezavantajları

- API gateway, ekstradan bir katman oluşturacağı için istek neticesinde işlevsel açıdan gözardı edilebilir bir farkla sürenin artmasına sebep olabilir.
- API gateway; geliştirici, dağıtım ve bakım gerektiren şahsına münhasır bir katmandır.

API Gateway

Dezavantajları

- Tüm servislere erişim api gateway üzerinden olduğu için herhangi bir çöküntü yahut kesinti durumunda tüm sistem aksaklığa uğrayabilir.

API Gateway - Ocelot

Açık kaynak olan **Ocelot**, API Gateway görevini yapar.



Oselot, kedigiller familyasının Leopardus cinsine ait türdür. Meksika, Güney Amerika ve Orta Amerika'da yaşayan bu türün uzunluğu kuyruğu ile birlikte 130 cm'e, ağırlıkları ise 8,2 kilogramdan 15,9 kilograma kadar ulaşabilir.

API Gateway - Ocelot

Ocelot, clienttan gelen istek neticesinde oluşturulan HttpRequest nesnesini arkaplandaki servislere iletir. Bunun için HttpRequestMessage nesnesi oluşturur ve bu şekilde isteği servislere ulaştırır.

Ocelot, uygulamanın pipeline'ındaki son ara katmandır.



API Gateway - Ocelot

Örnek senaryo :

Bunun için '**ProductAPI**' ve '**CostumerAPI**' olmak üzere iki proje oluşturalım.

```
dotnet new webapi --name productAPI
```

```
dotnet new webapi --name customerAPI
```

Şimdi bu projelere işlevsel özellik kazandırabilmek için sırasıyla '**ProductController**' ve '**CustomerController**' isimli controller sınıfları ekleyelim ve içeriklerini inşa edelim.

API Gateway - Ocelot

Örnek senaryo :

ProductAPI ;

```
[ApiController]  
[Route("api/[controller]")]  
public class ProductController : ControllerBase  
{  
    public IActionResult Get()  
    {  
        return Ok(new List<string> { "Kalem", "Kitap", "Silgi", "Defter" });  
    }  
}
```

API Gateway - Ocelot

Örnek senaryo :

CustomerAPI ;

```
[ApiController]  
[Route("api/[controller]")]  
public class CustomerController : ControllerBase  
{  
    public IActionResult Get()  
    {  
        return Ok(new List<string> { "Hilmi Celayir", "Saniye Yıldız", "Nevin Yıldız", "Fatih Yılmaz"  
        });  
    }  
}
```

API Gateway - Ocelot

Örnek senaryo :

Ardından her iki projeye localde çalışılacağından dolayı uygun bir port tanımlayalım.

Her iki proje içerisindeki '**launchSettings.json**' dosyalarındaki düzenlemeleri yapalım;

API Gateway - Ocelot

Örnek senaryo :

ProductAPI:

'launchSettings.json' :

```
"productAPI": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "launchUrl": "weatherforecast",  
  "applicationUrl": "https://localhost:5003;http://localhost:5002",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

API Gateway - Ocelot

Örnek senaryo :

CustomerAPI:

'launchSettings.json' :

```
"customerAPI": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "launchUrl": "weatherforecast",  
  "applicationUrl": "https://localhost:5001;http://localhost:5000",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

API Gateway - Ocelot

Örnek senaryo :

Microservislerimiz hazır!

Bundan sonra clientlardan gelecek olan istekleri tek elden karşılayabilecek ve isteğe uygun ilgili servise yönlendirecek olan api gateway projesini tasarlayalım.

API Gateway - Ocelot

Örnek senaryo :

Ocelot Entegrasyonu ve Konfigürasyonu

İlk olarak '**GatewayAPI**' isimli projemizi oluşturalım;

```
dotnet new webapi --name gatewayAPI
```

Gerekli düzenlemeleri yine 'launchSettings.json' dosyasında yapalım:

```
"getwayAPI": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "launchUrl": "weatherforecast",  
  "applicationUrl": "https://localhost:5005;http://localhost:5004",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

API Gateway - Ocelot

Örnek senaryo :

Ardından Ocelot kütüphanesini aşağıdaki kodu terminalde çalıştırarak ilgili projeye dahil edelim. ([Nuguet Ocelot](#))

dotnet add package Ocelot

Bu işlemten sonra uygulamanın herhangi bir dizininde(genellikle root tercih edilir) yapılandırmayı konfigüre edebilmek için '**ocelot.json**' isminde bir dosya oluşturalım ve içeriğini temel ve en sade biçimde olacak şekilde inşa edelim.

Bu kullanım Ocelot'a dair herhangi bir aktivite sergilemese de ilgili kütüphaneyi aktifleştirmek için yeterli olacaktır.

API Gateway - Ocelot

Örnek senaryo :

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/customer",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/api/gateway/custome",
      "UpstreamHttpMethod": [ "Get" ]
    },
  ],
}
```

```
{
  "DownstreamPathTemplate": "/api/product",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    {
      "Host": "localhost",
      "Port": 5003
    }
  ],
  "UpstreamPathTemplate": "/api/gateway/product",
  "UpstreamHttpMethod": [ "Get" ]
},
{
  "GlobalConfiguration": {
    "BaseUrl": "https://localhost:5005"
  }
}
```

API Gateway - Ocelot

Örnek senaryo :

DownstreamPathTemplate

Yönlendirme yapılacak microservice'in route'unu tutmaktadır.

DownstreamScheme

İlgili microservice'e yapılacak isteğin hangi protokol üzerinden gerçekleştirileceğini bildirmektedir.

DownstreamHostAndPorts

Microservice'in 'Host' ve 'Port' bilgilerini tutmaktadır.

UpstreamPathTemplate

API Gateway üzerinden microservice'e yapılacak yönlendirmenin route'unu tutmaktadır.

UpstreamHttpMethod

Hangi isteklerin yapılabileceği bildirilmektedir.

API Gateway - Ocelot

Örnek senaryo :

'ocelot.json' dosyasını 'program.cs' dosyası üzerinden uygulamaya dahil ediyoruz.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hosting, config) =>
        {
            config.AddJsonFile("ocelot.json", false, true);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```


API Gateway - Ocelot

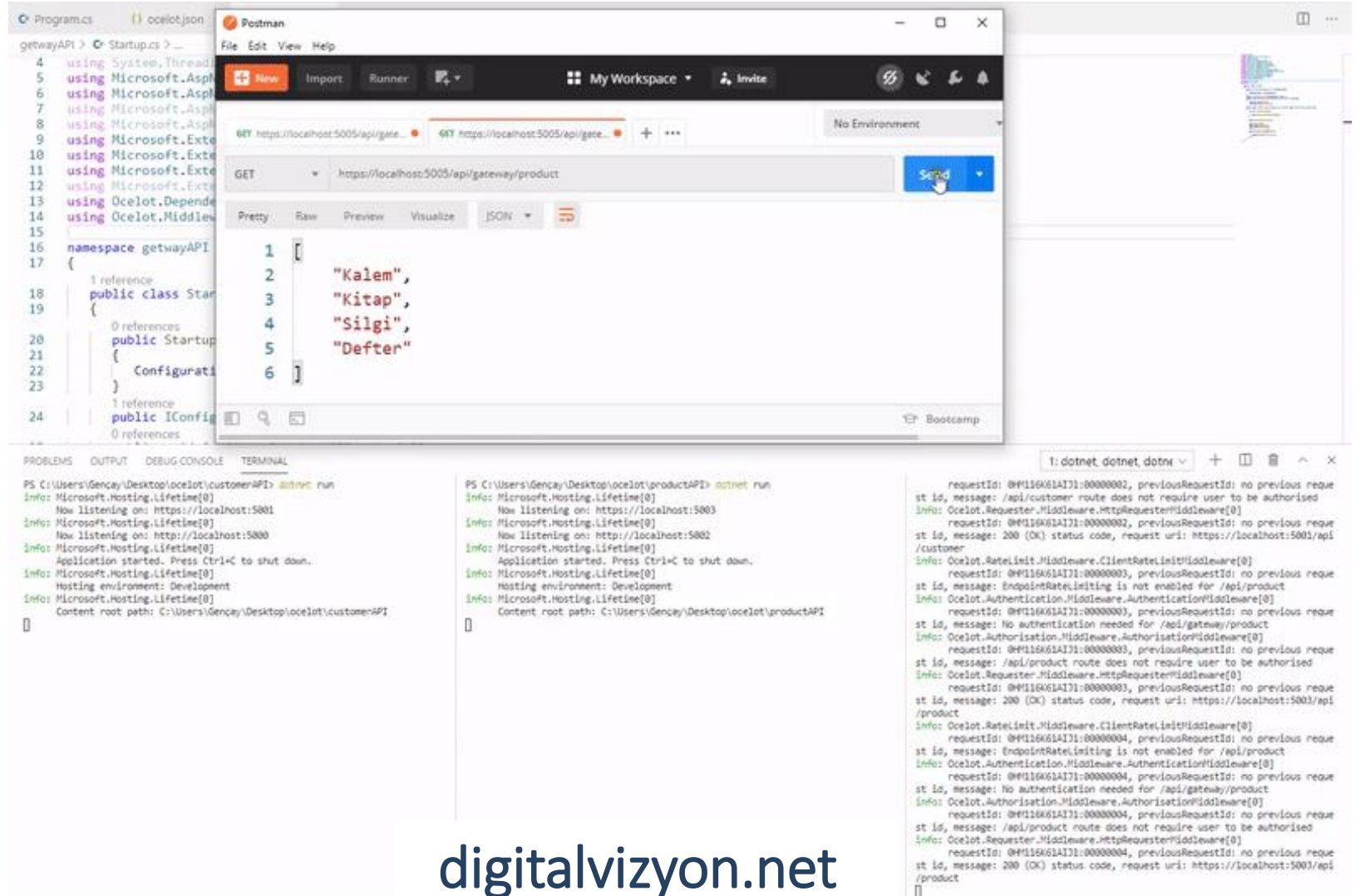
Örnek senaryo :

'startup.cs' dosyasında da programatik olarak **servis entegrasyonunu** ve **middleware** çağrısını gerçekleştiriyoruz;

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddOcelot();  
}  
async public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
    await app.UseOcelot();  
}
```

API Gateway - Ocelot

Örnek senaryo :



The screenshot displays a development environment for an Ocelot API Gateway. On the left, the Visual Studio Code editor shows the `Program.cs` file for the `gatewayAPI` project. The code includes necessary namespaces and defines a `Startup` class with a `Configure` method. The `Configure` method sets up the Ocelot configuration, including the `ProductAPI` endpoint.

In the center, the Postman client is open, showing a GET request to `https://localhost:5005/api/gateway/product`. The response is a JSON array containing the product details: `["Kalem", "Kitap", "Silgi", "Defter"]`.

At the bottom, the terminal window shows the output of the `dotnet run` command for both `customerAPI` and `productAPI` services. The logs indicate that both services are running successfully on their respective ports (5001 and 5002) and are listening for requests.

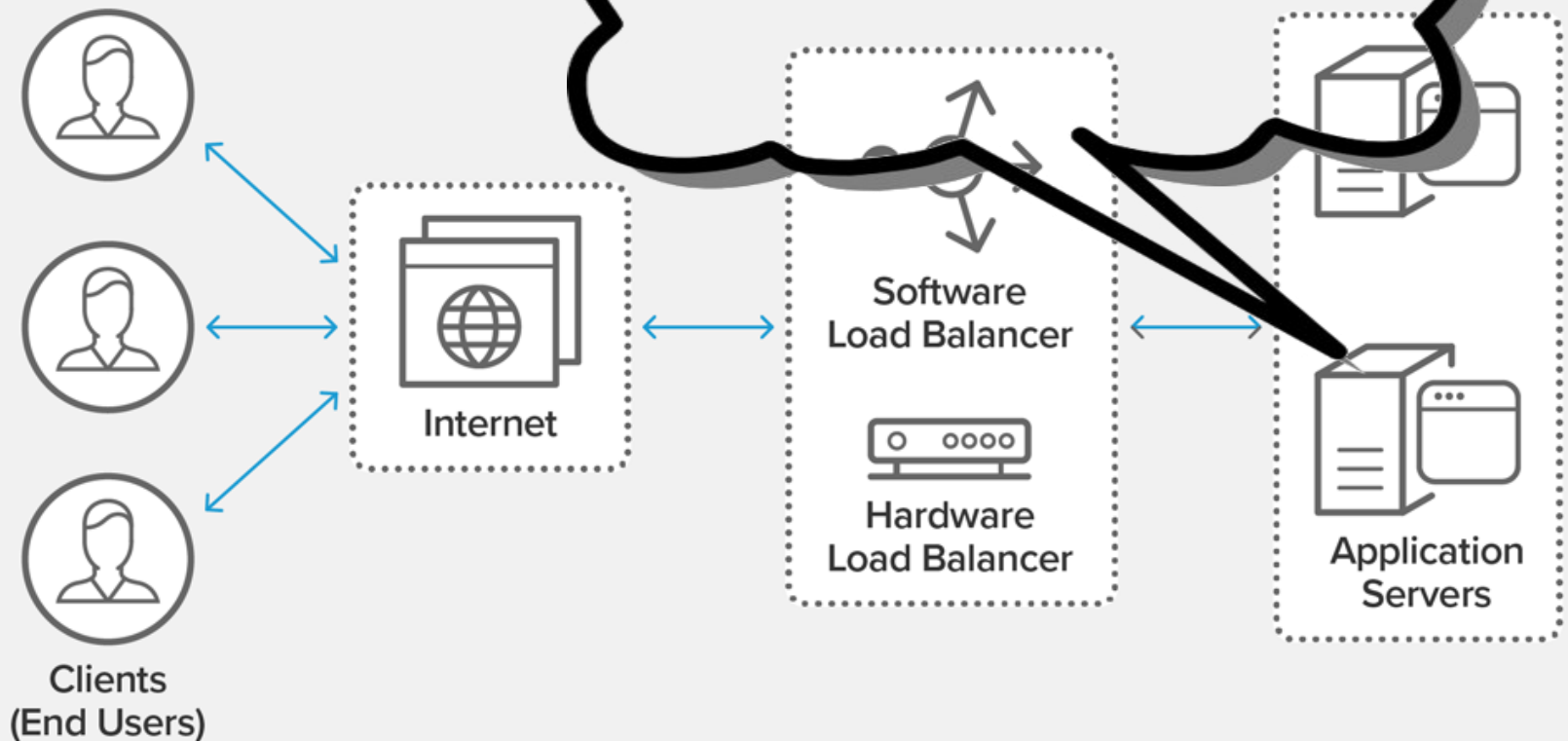
```
PS C:\Users\Gency\Deskto\ocelot\customerAPI> dotnet run
Info: Microsoft.Hosting.Lifetime[0]
Now listening on: https://localhost:5001
Info: Microsoft.Hosting.Lifetime[0]
Now listening on: http://localhost:5000
Info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
Info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
Info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\Gency\Deskto\ocelot\customerAPI

PS C:\Users\Gency\Deskto\ocelot\productAPI> dotnet run
Info: Microsoft.Hosting.Lifetime[0]
Now listening on: https://localhost:5003
Info: Microsoft.Hosting.Lifetime[0]
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\Gency\Deskto\ocelot\productAPI
```

```
requestId: @P116661A171:00000002, previousRequestId: no previous request id, message: /api/customer route does not require user to be authorised
Info: Ocelot.RequesterMiddleware.HttpRequestMiddleware[0]
requestId: @P116661A171:00000002, previousRequestId: no previous request id, message: 200 (OK) status code, request uri: https://localhost:5003/api/customer
Info: Ocelot.RateLimitMiddleware.ClientRateLimitMiddleware[0]
requestId: @P116661A171:00000003, previousRequestId: no previous request id, message: EndpointRateLimiting is not enabled for /api/product
Info: Ocelot.AuthenticationMiddleware.AuthenticationMiddleware[0]
requestId: @P116661A171:00000003, previousRequestId: no previous request id, message: No authentication needed for /api/gateway/product
Info: Ocelot.AuthorizationMiddleware.AuthorizationMiddleware[0]
requestId: @P116661A171:00000003, previousRequestId: no previous request id, message: /api/product route does not require user to be authorised
Info: Ocelot.RequesterMiddleware.HttpRequestMiddleware[0]
requestId: @P116661A171:00000003, previousRequestId: no previous request id, message: 200 (OK) status code, request uri: https://localhost:5003/api/product
Info: Ocelot.RateLimitMiddleware.ClientRateLimitMiddleware[0]
requestId: @P116661A171:00000004, previousRequestId: no previous request id, message: EndpointRateLimiting is not enabled for /api/product
Info: Ocelot.AuthenticationMiddleware.AuthenticationMiddleware[0]
requestId: @P116661A171:00000004, previousRequestId: no previous request id, message: No authentication needed for /api/gateway/product
Info: Ocelot.AuthorizationMiddleware.AuthorizationMiddleware[0]
requestId: @P116661A171:00000004, previousRequestId: no previous request id, message: /api/product route does not require user to be authorised
Info: Ocelot.RequesterMiddleware.HttpRequestMiddleware[0]
requestId: @P116661A171:00000004, previousRequestId: no previous request id, message: 200 (OK) status code, request uri: https://localhost:5003/api/product
```

API Gateway – Load Balancing

*- Ben yoğunum ablacım,
arkadaş ilgilensin 😊*



API Gateway – Load Balancing

Yük dengeleme, sunucu mimarilerindeki sınırlı sistem kaynaklarına rağmen artan yükün/trafiğin/isteğin karşılanabilmesi için sunucu üzerinde mevcut kaynakların donanımsal olarak arttırılmasına nazaran, sisteme aynı özellikte yeni sunucuların dahil olmasına ve bu birden fazla sunucu arasında kaynakların **orantılı** bir şekilde kullanılabileceği vaziyette trafiğin dağıtılmasına **Load Balancing** denir.

API Gateway – Load Balancing

Geleneksel yöntemlerde, gelen ve sistem kaynaklarını tüketen yoğun trafik/istek durumlarına karşı önlem amaçlı donanımsal kaynak arttırımı yöntemi tercih edilmektedir. Bu yöneme halk dilinde çoğaltmak, yükseltmek yahut arttırmak anlamına gelen **Scale-Up** ya da bir başka deyişle **Vertical Scaling(Dikey Ölçeklendirme)** denmektedir.

Scale Up



API Gateway – Load Balancing

Avantajları;

- Var olan sunucu üzerinde geliştirme yapıldığı için sunucu sayısı artmayacaktır. Haliyle bu durum enerji tasarrufunu getirecektir,
- Yukarıdaki sebepten dolayı sunucu sayısı sabit olacağından dolayı soğutma maliyetide daha az olacaktır,
- Teknik açıdan uygulaması kolaydır,
- Lisans maliyeti azdır.

API Gateway – Load Balancing

Dezavantajları;

- Sistemin yeri gelecek CPU, yeri gelecek RAM vs. gibi donanımsal ihtiyaçları olacağından dolayı maliyeti fazladır,
- Donanımsal arızalardan dolayı kesinti yaşanma ihtimali oldukça yüksektir. Ve bu durum tehlikeli bir risk teşkil etmektedir. Load balancing özellikle bu riske karşı verimli bir çözüm olarak sunulabilmektedir,
- Upgrade/yükseltme zorluğu vardır.

API Gateway – Load Balancing

Günümüzde sistemdeki mevcut sunucunun niceliksel olarak artırılması tercih edilmektedir. Bu yönetime de **Horizontal Scaling**(Yatay Ölçeklendirme) nispetinde **Scale-Out** denmektedir.

Scale Out



API Gateway – Load Balancing

Burada amacımız, gelen istek trafiğinin orantılı bir şekilde bu sunucular arasında paylaşımını sağlayarak yoğunluğu ölçeklendirebilmek ve böylece tek bir uygulama instance'ına tüm sorumluluğu vermeksizin yükü dengelemektir.

Birbirinin tekrarı olan sunucularda ayağa kaldırılmış birden fazla uygulama instance'ının gelen istek yoğunluğunu paylaşarak yükü dengelemesine **Load Balancing** denmektedir.

API Gateway – Load Balancing

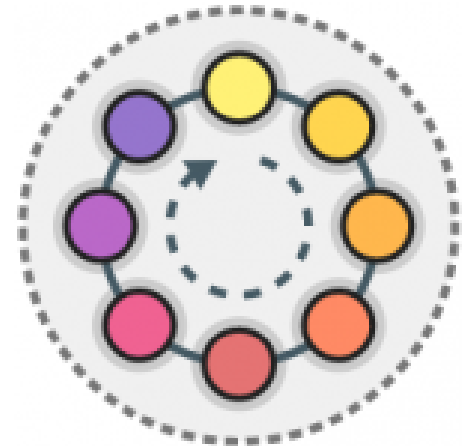
Kısaca, **sunucu mimarilerindeki** sınırlı sistem kaynaklarına rağmen artan yükün/trafiğin/isteğin karşılanabilmesi için sunucu üzerinde **mevcut kaynakların donanımsal olarak arttırılmasına nazaran**, **sisteme aynı özellikte yeni sunucuların dahil olmasına** ve bu birden fazla **sunucular arasında kaynakların orantılı** bir şekilde kullanılabileceği vaziyette **trafiğin dağıtılmasıdır yük dengeleme**.

API Gateway – Load Balancing

Load Balancing Algoritmaları;

Round Robin:

Gelen istekleri sunuculara sırasıyla dağıtan bir algoritmadır. Load balancer, dağıtım işlemi esnasında isteğin gönderildiği sunucuyu tutmakta ve sonraki istekleri itere ederek devam etmektedir. Sonuncu sunucuya gelindiği taktirde tekrardan başa döner ve istekleri aynı sırayla yönlendirmeye devam eder.

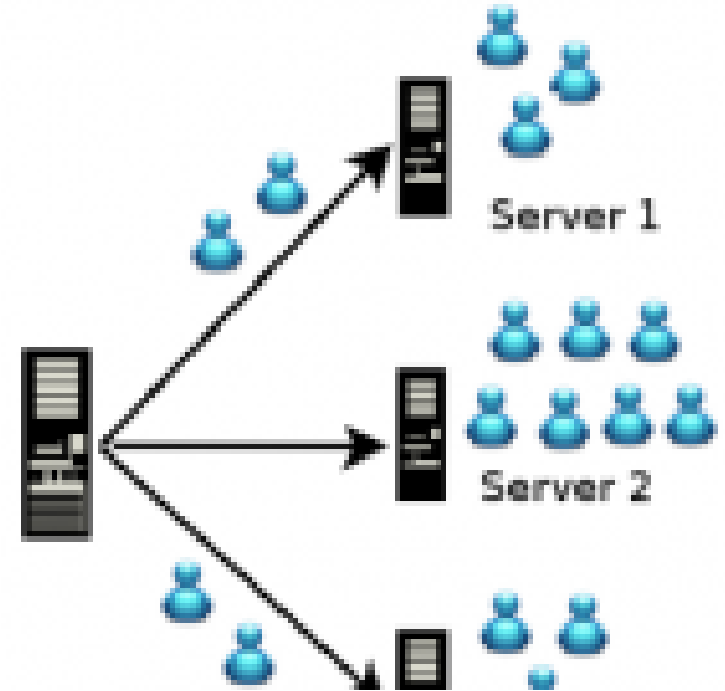


API Gateway – Load Balancing

Load Balancing Algoritmaları;

Least Connection:

Load balancer, gelen isteği en az yoğunlukta ve aktif olan sunucuya yönlendirir. Haliyle her sunucunun göreceli işlem kapasitesi mevcut olduğundan dolayı bu algoritma, istek neticesinde ağır ve uzun operasyonların yapılacağı durumlar için önerilmektedir.

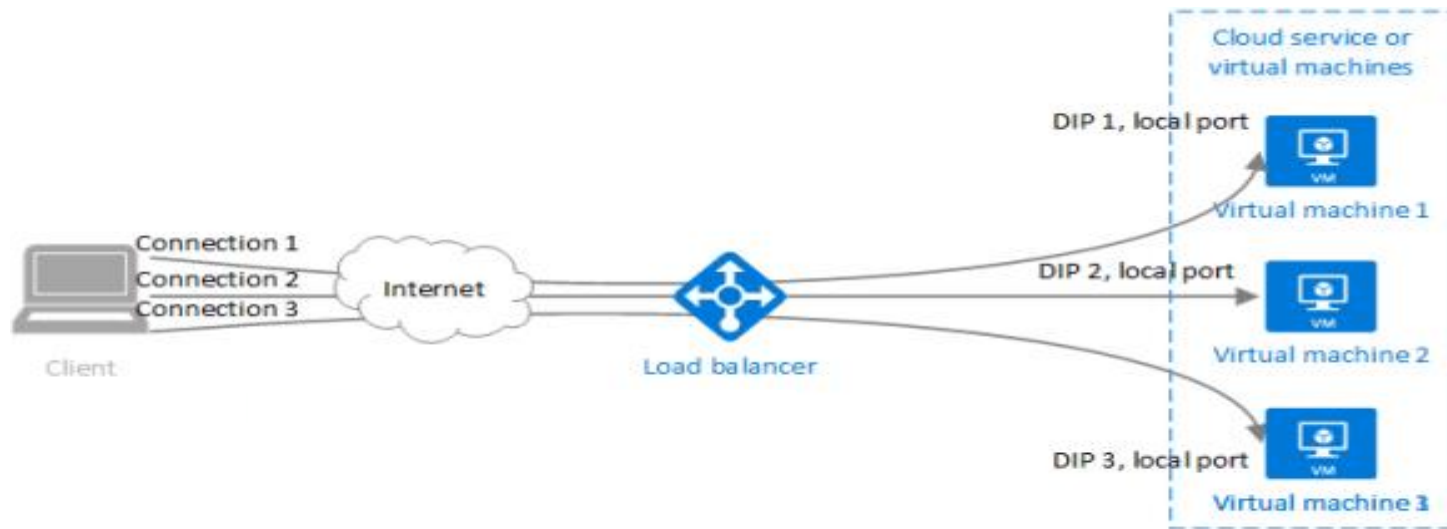


API Gateway – Load Balancing

Load Balancing Algoritmaları;

Source (IP Hash) :

Source algoritması, gelen isteğin IP değerine göre hangi sunucuda işleneceğini belirlememizi sağlamaktadır. Böylece her sunucu aynı IP'den gelen istekleri işleyecektir. Amaç sadece salt bölmedir. Genellikle istemcilerin öngörülebilir işlevsel hacimlerini sunuculara dağıtabilmek ve böylece her bir sunucuda farklı işlevleri icra edebilmek için tercih edilir.



API Gateway – Load Balancing

.NET CORE API UYGULAMASININ GELİŞTİRİLMESİ VE DOCKER'DA DEPLOY EDİLMESİ

Önce aşağıdaki gibi basit bir API geliştirelim;

```
[ApiController]
[Route("[controller]")]
public class ExampleController : ControllerBase
{
    readonly IConfiguration _configuration;
    public ExampleController(IConfiguration configuration)
    {
        _configuration = configuration;
    }
    [HttpGet]
    public IActionResult Get()
    {
        var data = _configuration["data"]; return Ok(data);
    }
}
```

API Gateway – Load Balancing

Burada tek dikkat edilmesi gereken husus, 'Configuration'dan 'data' key'ine karşılık, beklenen değerin geriye gönderilmesidir. Bu key'e karşılık değeri ayağa kaldırılacak container'a [environment](#) olarak göndereceğiz. Böylece 'Get' action'ına yapılan request neticesinde, load balancing ile hangi instance'a yönlendirildiğimizi rahatlıkla görebilmiş olacağız.

API'ı geliştirdikten sonra yapılması gereken husus Docker işlemleridir. Bunun için öncelikle ilgili API projesine bir Dockerfile dosyası eklemek ve içeriğini aşağıdaki gibi doldurmak gerekmektedir.

API Gateway – Load Balancing

FROM mcr.microsoft.com/dotnet/sdk:6.0

WORKDIR /app

COPY . .

COPY ["../Entity/Entity.csproj", "Entity/"]

COPY ["../Business/Business.csproj", "Business/"]

COPY ["../DAO/DAO.csproj", "DAO/"]

RUN dotnet restore

RUN dotnet publish WebAPI/WebAPI.csproj -c Release -o out

WORKDIR out

ENV ASPNETCORE_URLS=" https://:1000 "*

ENTRYPOINT ["dotnet", "WebAPI.dll"]

API Gateway – Load Balancing

Ardından powershell yahut cmd üzerinden;

docker build -f WebAPI\Dockerfile --force-rm -t customerimage .

talimatını vererek 'loadbalancerexample' isminde bir image oluşturulması gerekmektedir.

API Gateway – Load Balancing

Bu işlemten sonra son olarak container oluşturulması gerekmektedir.

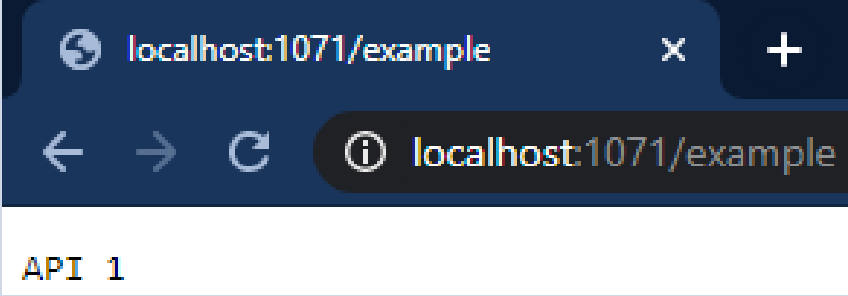
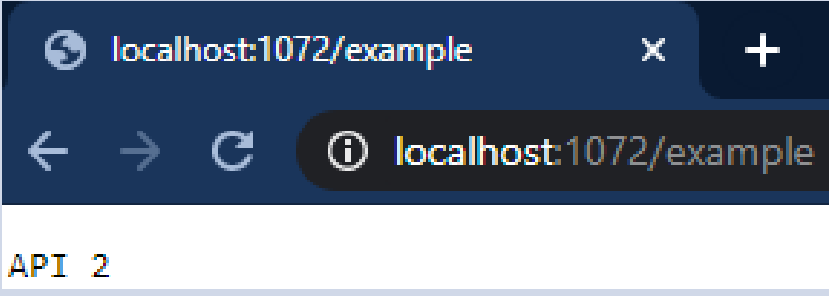
Bunun içinde yine powershell ya da cmd üzerinden aşağıdaki iki talimatı vererek 1071 ve 1072 portlarında olmak üzere iki container'ı ayağa kaldıralım.

```
docker run -p 2001:1000 --env data='API 1' --name api1 loadbalancerexample
```

```
docker run -p 2002:1000 --env data='API 2' --name api2 loadbalancerexample
```

API Gateway – Load Balancing

Şimdi ilgili portlar üzerinden yapılan istekler neticesinde nasıl bir sonuçla karşılaşıyoruz, inceleyelim.

Container 1	Container 2
	

API Gateway – Load Balancing

OCELOT İLE LOAD BALANCING OPERASYONU :

Ocelot ile load balancing operasyonunu gerçekleştirebilmek için bir API Gateway görevi görecektir. .NET Core temelli uygulama oluşturulmalıdır.

Ardından bu uygulamamızda yine [Nuget'ten Ocelot](#) kütüphanesinin yüklenmesi gerekmektedir.

Devamında ise uygulamanın ana dizinine 'ocelot.json' isminde bir dosya ekleyerek içeriğini aşağıdaki gibi doldurmak gerekmektedir.

API Gateway – Load Balancing

ocelot.json

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/example",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 1071
        },
        {
          "Host": "localhost",
          "Port": 1072
        }
      ],
      "UpstreamPathTemplate": "/example",
      "LoadBalancerOptions": {
        "Type": "LeastConnection"
      },
      "UpstreamHttpMethod": [ "Get" ]
    },
    {
      "GlobalConfiguration": {
        "BaseUrl": "https://localhost:5001"
      }
    }
  ]
}
```

API Gateway – Load Balancing

Yukarıdaki yapılanmayı parça parça incelersek eğer;

4. satır; *DownstreamPathTemplate* API uygulamasının route bilgisi bildirilmektedir.

5. satır; *DownstreamScheme* İsteklerin hangi protokol üzerinden gerçekleştirileceği bildirilmektedir.

6. satır; *DownstreamHostAndPorts* Load balancing operasyonunun yapılacağı Host ve Port bilgileri verilmektedir.

16. satır; *UpstreamPathTemplate* API Gateway görevi gören bu uygulamaya hangi route üzerinden istek atılacağını belirlemektedir.

17. satır; *LoadBalancerOptions* Load balancer konfigürasyonları belirlenmektedir.

18. satır; *Type* Hangi load balancing algoritmasının kullanılacağı bildirilmektedir.

Ocelot; ***LeastConnection, RoundRobin, NoLoadBalancer ve CookieStickySessions*** olmak üzere dört farklı algoritma kullanmaktadır.

API Gateway – Load Balancing

LeastConnection

Gelen isteği en az maliyette çalışan ya da başka bir deyişle en az hizmet yoğunluğunda olan sunucuya yönlendirir.

RoundRobin

Gelen isteği sunucular arasında döngüsel olarak sırasıyla paylaştırır.

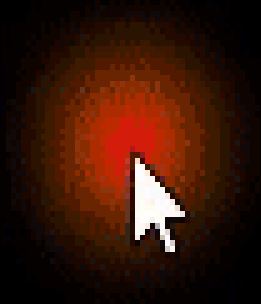
NoLoadBalancer

Gelen isteğin nereye yönlendirileceğini Service Discovery'den alır. **Service Discovery**, servislerin birbirleriyle iletişim kurabilmesini, health check ile sadece ayakta olan servislerin kullanılabilmesini ve load balancing ile servislerin dinamik olarak dağılmasını sağlayan bir özelliktir.

CookieStickySessions

Bir kullanıcıdan gelen tüm istekleri sürekli aynı sunucuya yönlendirir.

API Gateway – Load Balancing



digitalvizyon.net

API Gateway – Load Balancing

Assign static IP to Docker container

İlk önce kendi ağımızı oluşturalım,

- *docker network create --subnet=172.18.0.0/16 mynet123*

Sonra oluşturduğumuz ağ üzerinde image imizi çalıştıralım..

- *docker run --net mynet123 --ip 172.18.0.22 -p 1072:1453 --env data='API 1' --env delay=1000 --name api1 loadbalancerexample*

--*hostname* to specify a hostname

--add-host to add more entries to /etc/hosts

API Gateway – RabbitMQ

Docker’da RabbitMQ

Ön Hazırlık

- Bilgisayarınıza Docker kurulumunu gerçekleştirin.
- Docker’ı başlatın.
- Powershell açın ve aşağıdaki kodu yazıp, çalıştırın.

`docker –version`

Hata almadan Docker sürümünü görüyorsanız başarılı bir şekilde yükleme ve çalıştırmayı gerçekleştirmişsiniz demektir.

API Gateway – RabbitMQ

RabbitMQ Image'inin İndirilmesi

Herşeyden önce **RabbitMQ image** 'ının Docker'a indirilmesi gerekmektedir.

Bunun için aşağıdaki komutu powershell'de çalıştırmanız yeterlidir.

`docker pull rabbitmq`

API Gateway – RabbitMQ

RabbitMQ Docker Container Oluşturma

İndirilen RabbitMQ image 'ndan bir **container** ayağa kaldıralım

```
docker run -d -p 15672:15672 -p 5672:5672 --name rabbitmqcontainer rabbitmq:3-management
```

API Gateway – RabbitMQ

RabbitMQ Docker Container Oluşturma

```
docker run -d -p 15672:15672 -p 5672:5672 --name rabbitmqcontainer rabbitmq:3-management
```

Yukarıdaki komutu powershell'de çalıştırdığınızda RabbitMQ message broker'ın Docker'da çalıştığı 5672 portunu bilgisayarımızdaki 5672 portuna bağlıyoruz.

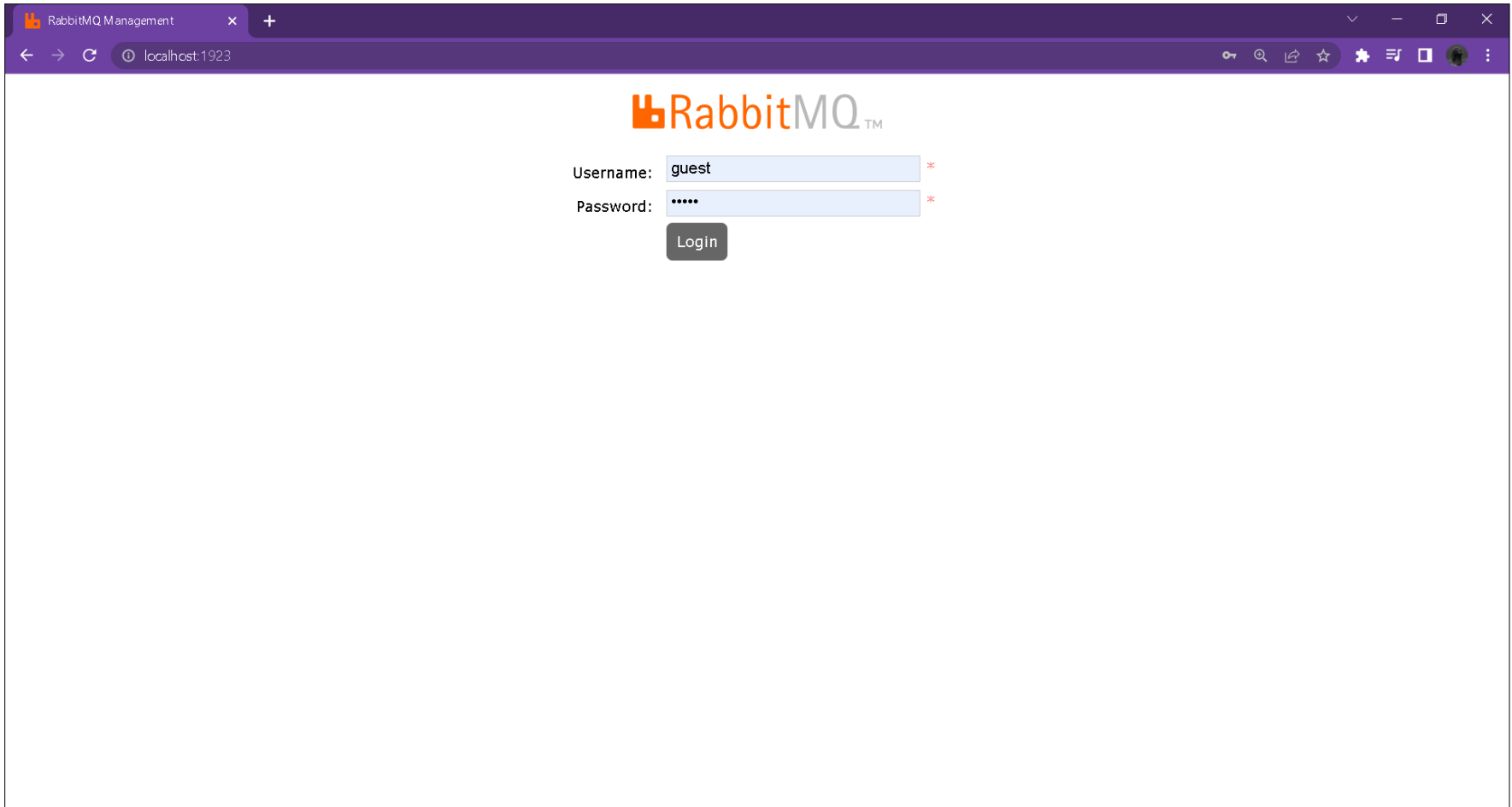
Benzer şekilde localhost için 15672 portunda yine bilgisayarımızdaki 15672 portuna bağlayarak container'ı ayağa kaldırıyoruz.

API Gateway – RabbitMQ

RabbitMQ Docker Container Oluşturma

Bu işlemden sonra herhangi bir web tarayıcısına <http://localhost:15672> adresini yazıp enter'a basmanız yeterlidir.


API Gateway – RabbitMQ



The screenshot shows the RabbitMQ Management web interface in a browser. The browser's address bar displays 'localhost:1923'. The page features the RabbitMQ logo at the top center. Below the logo, there is a login form with two input fields: 'Username' containing the text 'guest' and 'Password' containing six dots. Each input field has a small red asterisk to its right. A 'Login' button is positioned below the password field.

RabbitMQ Management

localhost:1923

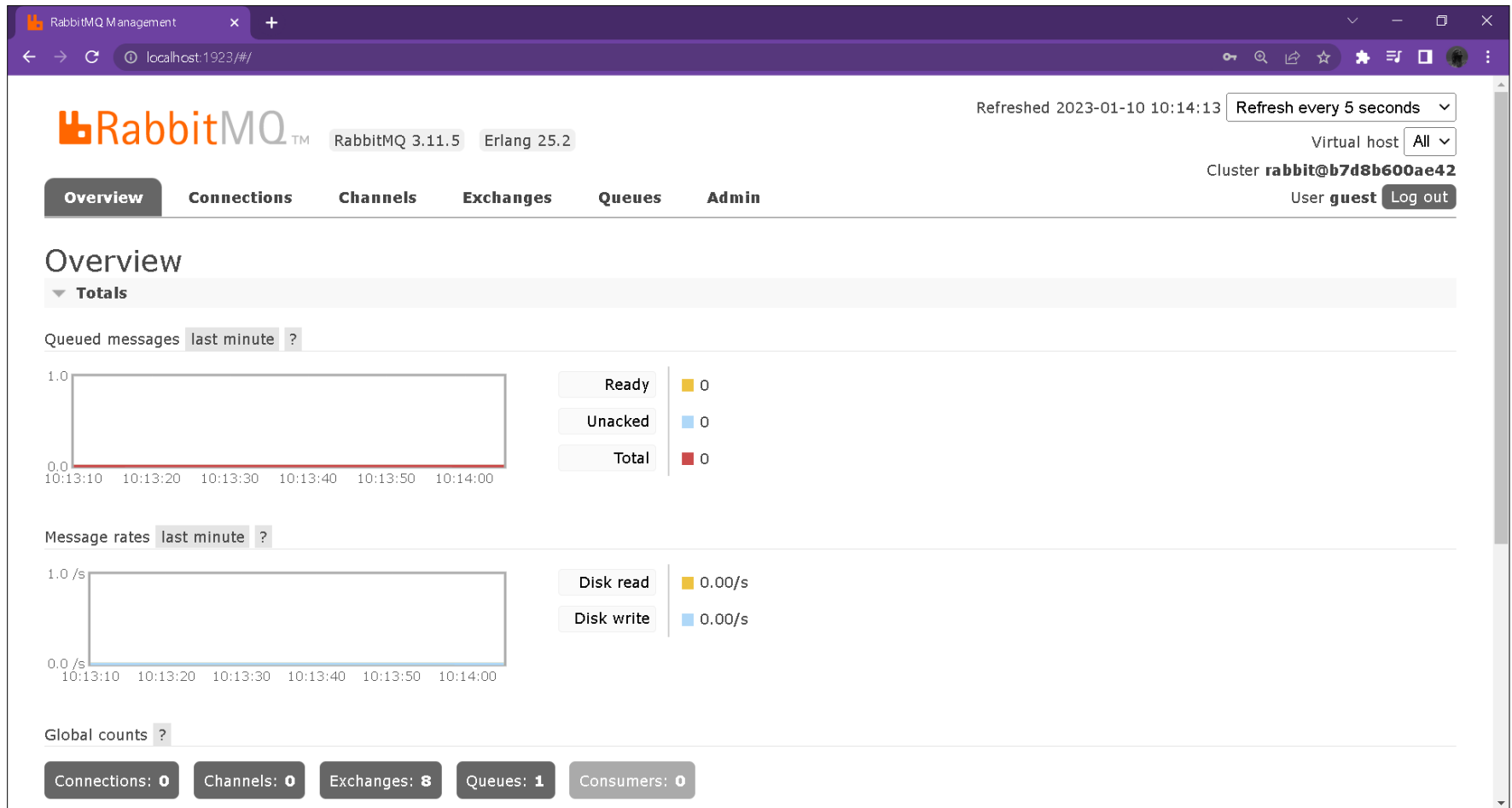
 RabbitMQ™

Username: *

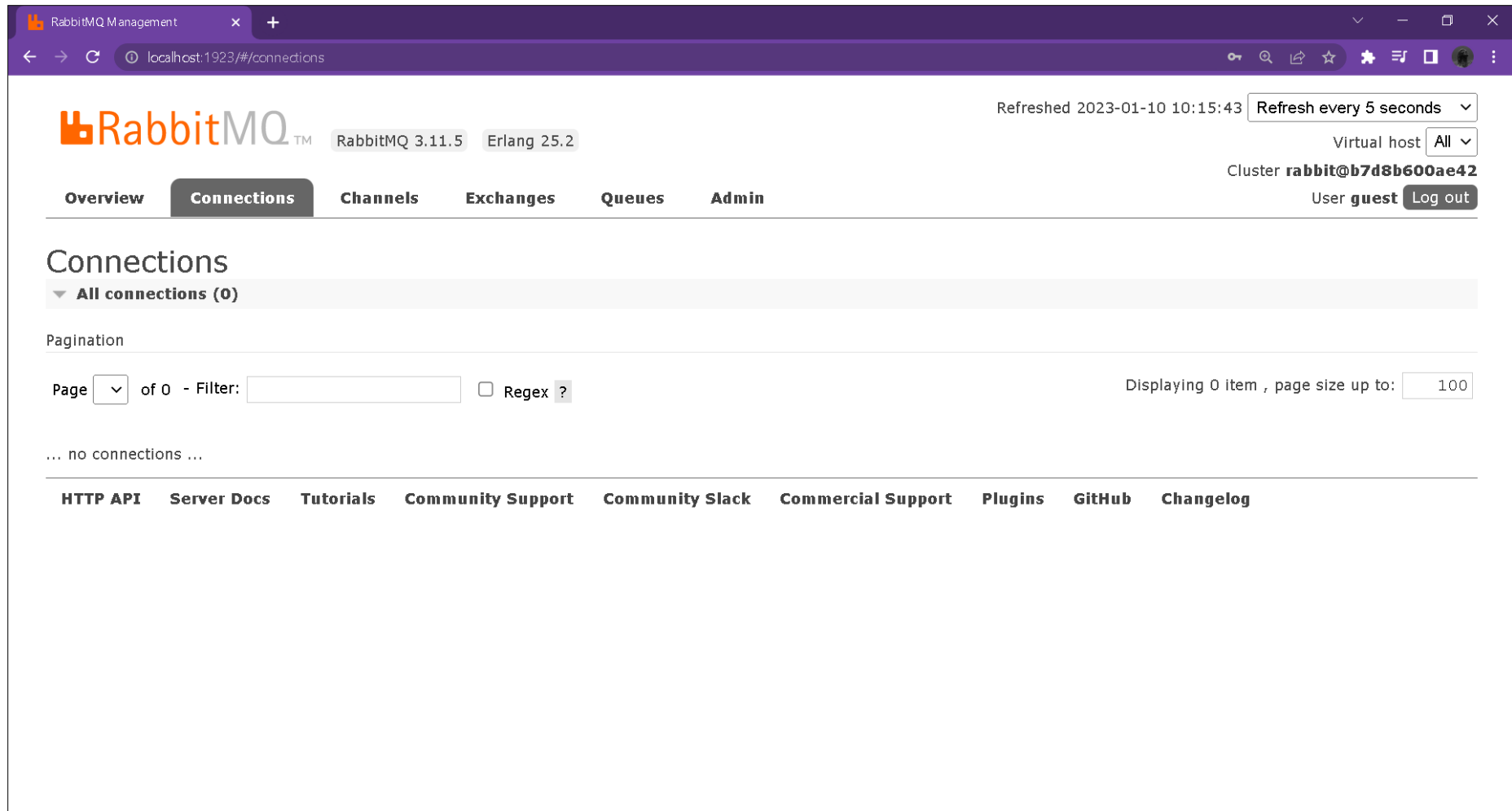
Password: *

Login

API Gateway – RabbitMQ



API Gateway – RabbitMQ



The screenshot shows the RabbitMQ Management interface in a web browser. The browser tab is titled "RabbitMQ Management" and the address bar shows "localhost:15672/#/connections". The page header includes the RabbitMQ logo, version information (RabbitMQ 3.11.5, Erlang 25.2), and a refresh button. The main navigation bar has tabs for Overview, Connections (selected), Channels, Exchanges, Queues, and Admin. The right sidebar shows the cluster name "rabbit@b7d8b600ae42", the user "guest", and a "Log out" button. The main content area is titled "Connections" and shows a filter for "All connections (0)". Below this, there is a pagination section with a "Page" dropdown, a filter input, a "Regex" checkbox, and a "Displaying 0 item, page size up to: 100" message. At the bottom, there is a footer with links for HTTP API, Server Docs, Tutorials, Community Support, Community Slack, Commercial Support, Plugins, GitHub, and Changelog.

RabbitMQ Management x +

localhost:15672/#/connections

Refreshed 2023-01-10 10:15:43 Refresh every 5 seconds

Virtual host All

Cluster rabbit@b7d8b600ae42

User guest Log out

Overview **Connections** Channels Exchanges Queues Admin

Connections

▼ All connections (0)

Pagination

Page ▼ of 0 - Filter: ☐ Regex ?

Displaying 0 item , page size up to: 100

... no connections ...

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

API Gateway – CAP

CAP Nedir? Nasıl Kullanılır?

CONSISTENCY - Tutarlılık

AVAILABILITY - Kullanılabilirlik

PARTITION RESISTANCE – Bölüm Direnci

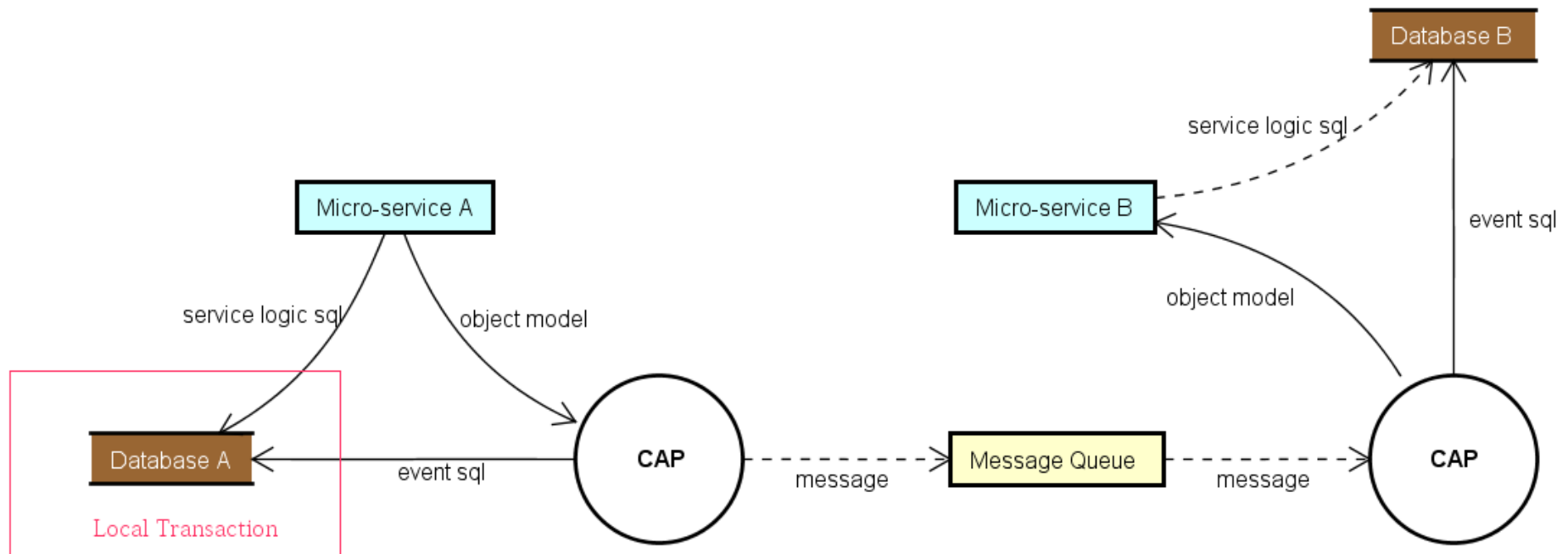
API Gateway – CAP

CAP Nedir? Nasıl Kullanılır?

CAP, distributed sistemlerdeki producer-consumer yapısının güvenilirliği tam olarak garanti etmemesi durumuna istinaden, yönetilebilirlik açısından kolaylık sağlayan ve Event Bus işlevine sahip olan, hafif, kolay ve verimli bir .NET tabanlı açık kaynak (open source) kütüphanedir.

API Gateway – CAP

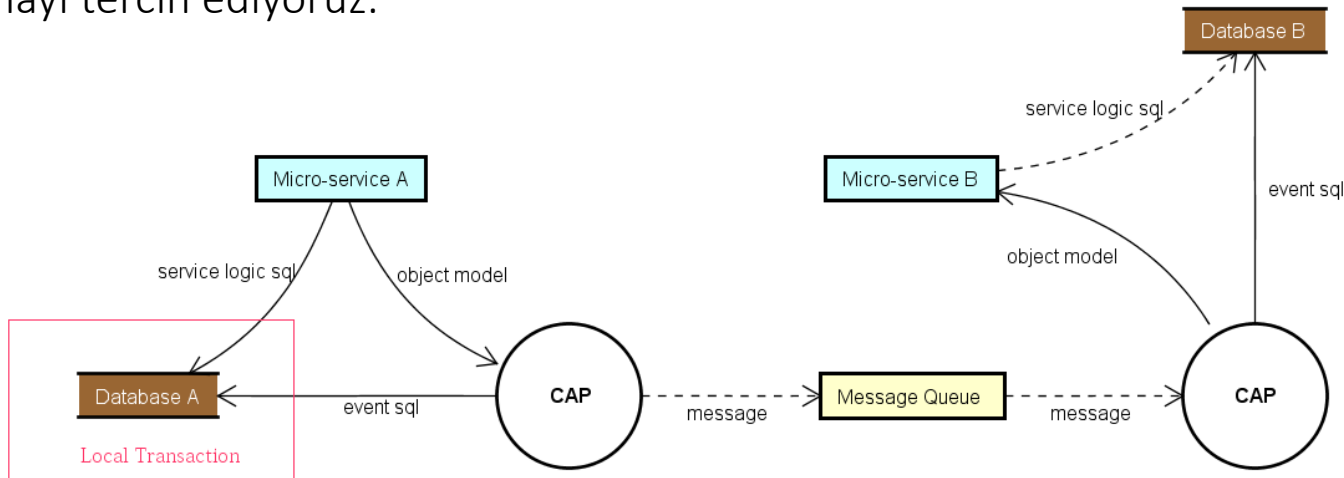
CAP Nedir? Nasıl Kullanılır?



API Gateway – CAP

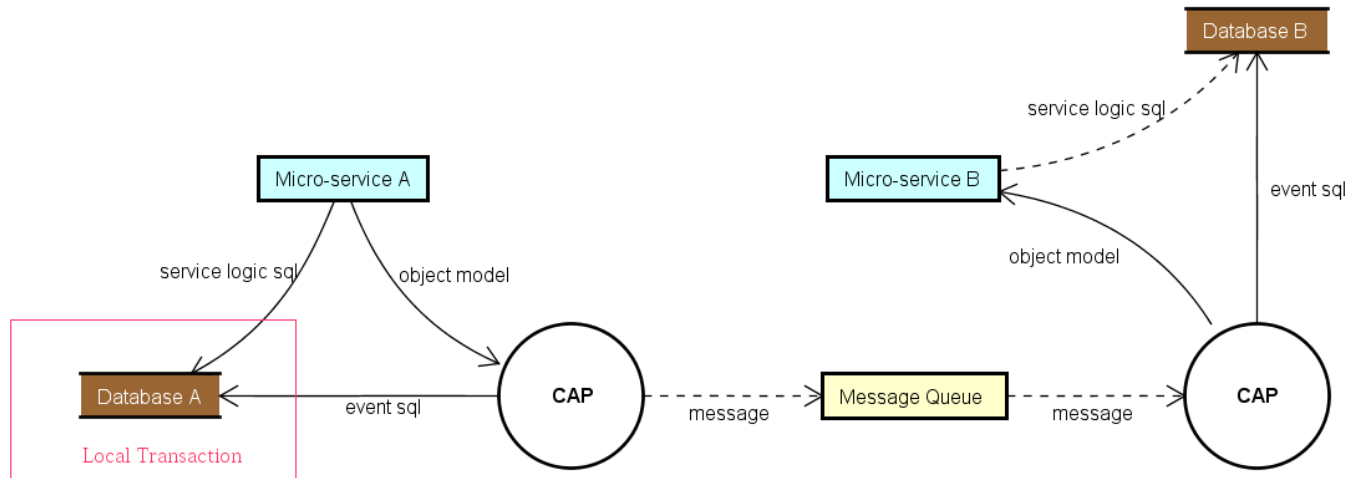
CAP Nedir? Nasıl Kullanılır?

Microservice yaklaşımını benimseyen sistemlerde mesaj kuyruk sisteminin basitçe kullanılması güvenilirliği garanti etmeyecektir. İşte bu olası duruma istinaden dağıtılmış sistemlerin birbirini çağırması sürecinde oluşabilecek istisnaları çözümllemek için CAP'ı kullanmayı tercih ediyoruz.



API Gateway – CAP

*“CAP, microservice mimarisinde servislerin birbirleriyle iletişime geçmesinde oluşacak istisnaları çözümleyebilmek için mevcut veritabanı ile entegre çalışan **Local Message Table** yaklaşımını benimsemiş bir kütüphanedir”*



API Gateway – Event Driven

Microservice’lerde Event Driven Nedir?

Her bir servisin işlevsel açıdan yapacağı operasyon neticesinde bir event’i fire ederek sonucu/değeri/neticeyi herhangi bir message queue(RabbitMQ, Kafka vs.) sistemine atarak bu kuyrukları dinleyen diğer servisleri uyarmasına event-driven denir.

API Gateway – Event Driven

Event Driven'ın Avantajları Nelerdir?

- Gevşek bağlı(loosely coupled) bir mimari oluşturulmasına olanak sağlar.
 - Servisler arasında asenkron bir iletişim sağlanacağı için performansı arttırır.
 - Servisler arasında mesajlaşma sistemleri kullanılacağından dolayı, event fire edildiği takdirde consumer servis'e bir sebepten dolayı erişilemezse eğer kuyruk verinin kalıcılığını sağlayacak ve tekrar erişilebilir olunca tüketim devam edecektir.
- Böylece süreçte veri kaybı yaşamaksızın sağlıklı işlevsellik söz konusu olacaktır.
- Ölçeklenebilirlik(scalability) sağlar.

API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

İlk olarak aralarında iletişim kuracak olan 'ProducerAPI' ve 'ConsumerAPI' isminde iki adet servis oluşturalım.

```
dotnet new webapi --name producerAPI
```

```
dotnet new webapi --name consumerAPI
```

API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

Her iki projeyede DotNetCore.CAP kütüphanesini

dotnet add package DotNetCore.CAP

komutuyla ekleyelim.

API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

Kullanacağımız RabbitMQ mesaj kuyruk sistemi içinse DotNetCore.CAP.RabbitMQ
Kütüphanesini

dotnet add package DotNetCore.CAP.RabbitMQ

komutuyla ekleyelim

API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

Son olarak servisler arasında iletişim için yaratılan olayları loglayabilmek için

DotNetCore.CAP.SqlServer kütüphanesini

dotnet add package DotNetCore.CAP.SqlServer

komutu aracılığıyla ekleyelim.

API Gateway – Event Driven

İlgili kütüphaneleri yükledikten sonra her iki projeninde 'Startup.cs' dosyasında aşağıdaki konfigürasyonları yapalım;

API Gateway – Event Driven

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ExampleContext>(options => options.UseSqlServer("Server=.;Database=Icisleri
DB;Trusted_Connection=True;"));
    services.AddCap(options =>
    {
        options.UseEntityFramework<ExampleContext>();
        options.UseSqlServer("Server=.;Database=IcisleriDB;Trusted_Connection=True;");

        options.UseRabbitMQ(options =>
        {
            options.ConnectionFactoryOptions = options =>
            {
                options.Ssl.Enabled = false;
                options.HostName = "localhost";
                options.UserName = "guest";
                options.Password = "guest";
                options.Port = 5672;
            };
        });
    });
    services.AddControllers();
}
```

API Gateway – Event Driven

Port Ayarlaması

Her iki projenin 'launchSettings.json' dosyasında aşağıdaki gibi port ayarlamasında Bulunalım.

API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

ProducerAPI;

```
"producerAPI": {  
  "commandName": "Project",  
  "launchBrowser": false,  
  "applicationUrl": "https://localhost:5001;http://localhost:5000",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```


API Gateway – Event Driven

Servislerin Oluşturulması ve CAP Entegrasyonu

ConsumerAPI;

```
"consumerAPI": {  
  "commandName": "Project",  
  "launchBrowser": false,  
  "applicationUrl": "https://localhost:5003;http://localhost:5002",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

API Gateway – Event Driven

Mesaj yayınlama ;

Şimdi ProducerAPI servisi üzerinden bir işlem gerçekleştirelim ve neticede ConsumerAPI servisini uyaralım.

Bunun için ProducerAPI uygulamasına bir controller yazalım;

API Gateway – Event Driven

```
[ApiController]
[Route("api/[controller]")]
public class ProducerController : ControllerBase
{
    private readonly ICapPublisher _capPublisher;
    public ProducerController(ICapPublisher capPublisher)
    {
        _capPublisher = capPublisher;
    }
    public async Task<ActionResult> ProducerTransaction()
    {
        using ExampleContext context = new ExampleContext();
        using var transaction = context.Database.BeginTransaction(_capPublisher, autoCommit:
true);
        var date = DateTime.Now;
        await _capPublisher.PublishAsync<DateTime>("producer.transaction", date);
        return Ok(date);
    }
}
```

API Gateway – Event Driven

Mesaj yayınlama ;

Burada 'ICapPublisher' interface'i ile CAP kütüphanesi inject edilmekte ve context nesnesi üzerinden başlatılan transactionda kullanılmaktadır.

Ayrıca ilgili arayüz aracılığıyla 'producer.transaction' değeri altında veri olarak o anın tarih bilgisini yayınlamaktadır.

API Gateway – Event Driven

Mesaj yakalama;

ProducerAPI tarafından yayınlanan mesajı ConsumerAPI’da elde edebilmek için ilgili projede de bir controller geliştirilmelidir;

API Gateway – Event Driven

```
[ApiController]  
[Route("[controller]")]  
public class ConsumerController : ControllerBase  
{  
    [CapSubscribe("producer.transaction")]  
    public void Consumer(DateTime date)  
    {  
        Console.WriteLine(date);  
    }  
}
```

Kod bloğunu incelendiğinde **consumer**'ın dinleyici fonksiyonunu '**CapSubscribe**' attribute'u ile işaretleyerek **producer**'de ki publish değeri olan 'producer.transaction' ile ilişkilendiriyor ve ilgili yayına abone yapıyoruz.

Böylece producer bir mesaj yayınladığı an direkt olarak consumer haberdar ediliyor.



API Gateway – Event Driven

Business Logic Service 'lerde Mesaja Yakalama

Mesajları “**controller**” 'larda olduğu gibi “**business logic**” 'de de yakalayabiliriz.

Bunun için ilgili sınıf içerisinde birtakım geliştirmeler yapmak yeterli olacaktır.

API Gateway – Event Driven

Business Logic Service 'lerde Mesaja Yakalama

Mesajları “**controller**” ‘larda olduğu gibi “**business logic**” ‘de de yakalayabiliriz.

Bunun için ilgili sınıf içerisinde birtakım geliştirmeler yapmak yeterli olacaktır.

```
public class ConsumerService : ICapSubscribe  
{  
    [CapSubscribe("producer.transaction")]  
    public void Consumer(DateTime date)  
    {  
        Console.WriteLine("Servis : " + date);  
    }  
}
```

```
Subscribe(producer.transaction);  
public void Consumer(DateTime date)  
{  
    Console.WriteLine("Servis : " + date);  
}
```

API Gateway – Event Driven

Business Logic Service 'lerde Mesaja Yakalama

Burada business logic sınıfı 'ICapSubscribe' interface'inden türemesi gerekmekte ve 'CapSubscribe' attribute'u ile tekrardan ilgili producer değerini taşıyacak şekilde işaretlenmesi gerekmektedir.

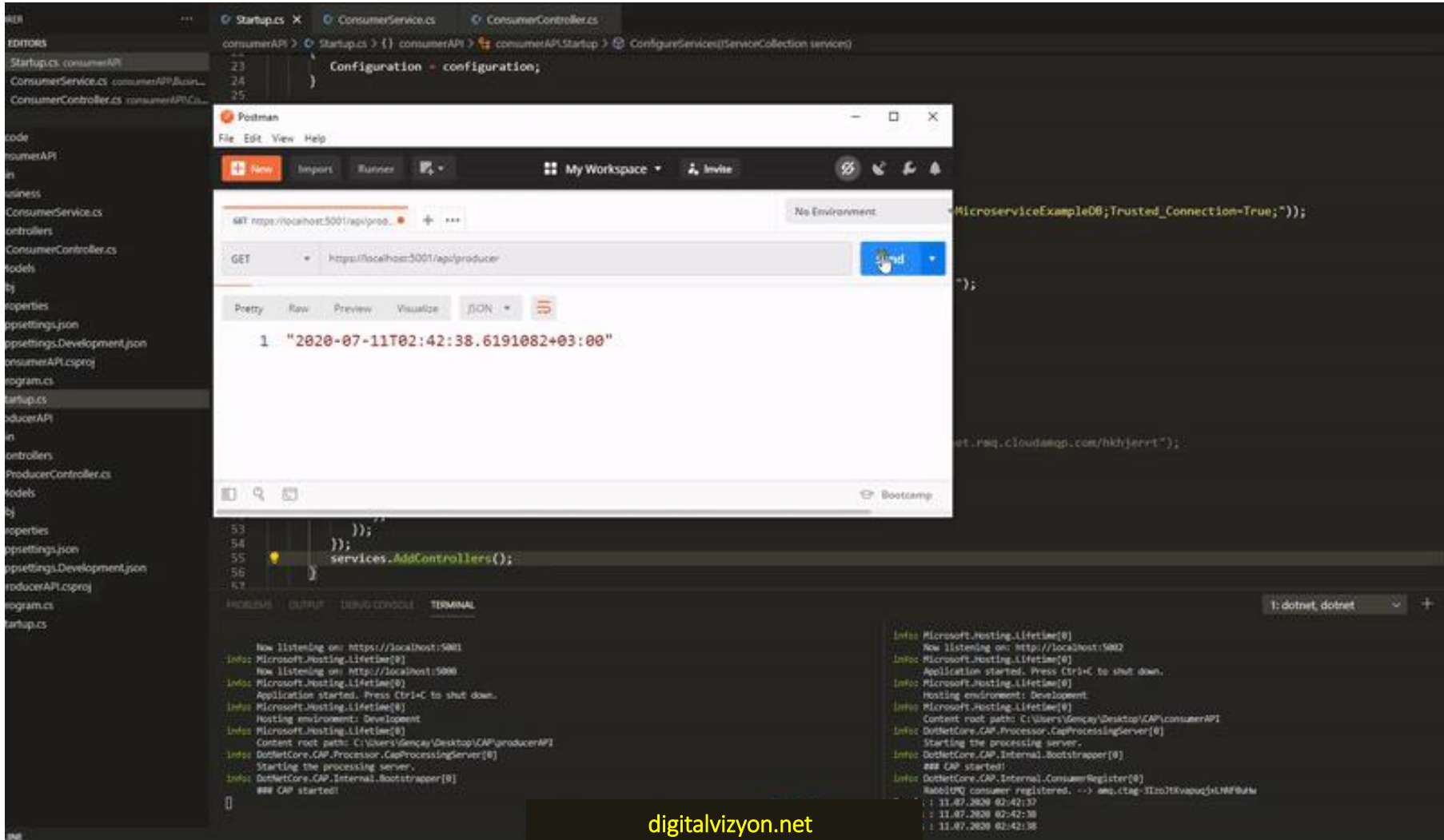
Bu işlemin ardından inşa edilen ilgili sınıf uygulamaya servis olarak eklenmelidir.

API Gateway – Event Driven

Business Logic Service 'lerde Mesaja Yakalama

```
public void ConfigureServices(IServiceCollection services)  
{  
    .  
    .  
    services.AddTransient<ConsumerService>();  
    services.AddCap(options =>  
    {  
        .  
        .  
    });  
    .  
    .  
}
```

API Gateway – Event Driven



The screenshot displays a Visual Studio Code editor with a .NET Core application. The **Postman** client is open, showing a **GET** request to `https://localhost:5001/api/producer`. The response is a JSON array containing one object: `1 { "2020-07-11T02:42:38.6191082+03:00" }`. The **Terminal** window at the bottom shows the application's startup logs, including the message `Now listening on: https://localhost:5001` and `Application started. Press Ctrl+C to shut down.` The **Code** editor shows the `Startup.cs` file with the `services.AddControllers();` line highlighted.

digitalvizyon.net

API Gateway – Event Driven

Abone Grupları Oluşturma

CAP, gelen bir mesajı birden fazla gruba dağıtarak daha fazla servis yahut controller tarafından dinlenilmesini sağlayabilmektedir.

API Gateway – Event Driven

Abone Grupları Oluşturma

Service;

```
[CapSubscribe("producer.transaction", Group = "group2")]  
public void Consumer(DateTime date)  
{  
    Console.WriteLine("Servis : " + date);  
}
```

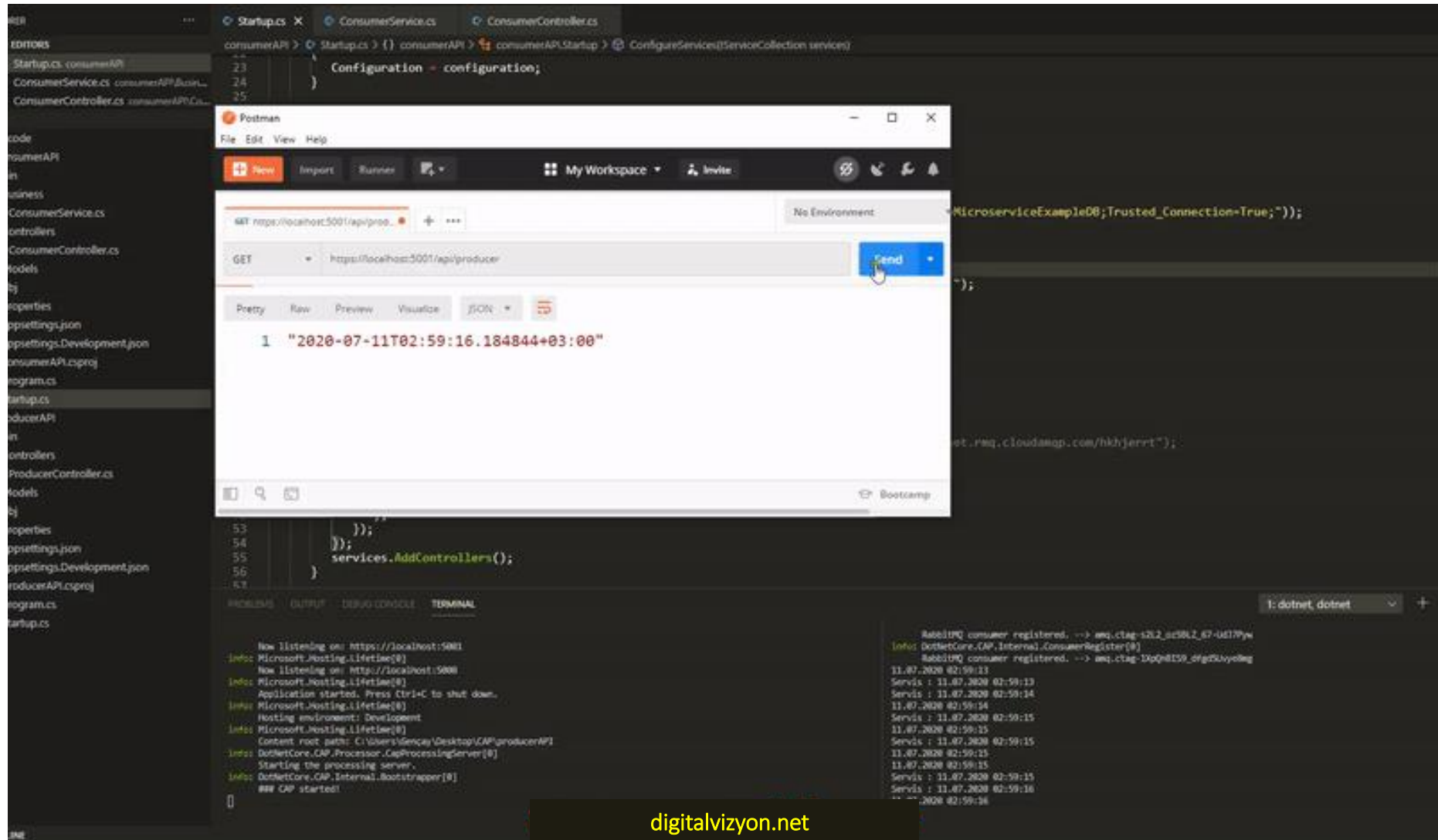
API Gateway – Event Driven

Abone Grupları Oluşturma

Controller;

```
[ApiController]  
[Route("[controller]")]  
public class ConsumerController : ControllerBase  
{  
    [CapSubscribe("producer.transaction", Group = "group1")]  
    public void Consumer(DateTime date)  
    {  
        Console.WriteLine(date);  
    }  
}
```

API Gateway – Event Driven



The screenshot displays a Visual Studio Code environment with a .NET Core API project. The file explorer on the left lists files such as `Startup.cs`, `ConsumerService.cs`, and `ConsumerController.cs`. The editor shows `Startup.cs` with a configuration section:

```
consumerAPI > Startup.cs > {} consumerAPI > consumerAPI.Startup > ConfigureServices(IServiceCollection services)  
Configuration = configuration;
```

A Postman window is open, showing a GET request to `http://localhost:5001/api/producer`. The response is a timestamp: `"2020-07-11T02:59:16.184844+03:00"`.

The bottom of the screen shows the terminal with application logs:

```
Now listening on: https://localhost:5001  
Info: Microsoft.Hosting.Lifetime[0]  
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\Genca\Desktop\CAP\producerAPI  
Info: DotNetCore.CAP.Processor.CapProcessingServer[0]  
Starting the processing server.  
Info: DotNetCore.CAP.Internal.Bootstrapper[0]  
## CAP started!
```


API Gateway – Event Driven

Gösterge Paneli Oluşturma(Dashboard)

CAP, gönderilen ve alınan mesajları kolayca gerçek zamanlı görebilmemiz için Dashboard özelliği taşımaktadır.

Dashboard entegrasyonu için uygulamaya [DotNetCore.CAP.Dashboard](#) kütüphanesini Eklememiz gerekmektedir.

```
dotnet add package DotNetCore.CAP.Dashboard
```

API Gateway – Event Driven

Gösterge Paneli Oluşturma(Dashboard)

Ardından 'Startup.cs' dosyasında aşağıdaki konfigürasyonu gerçekleştiriniz.

```
public void ConfigureServices(IServiceCollection services)  
{  
    .  
    services.AddCap(options =>  
    {  
        .  
        .  
        options.UseDashboard(o => o.PathMatch = "/cap-dashboard");  
        .  
        .  
    });  
    .  
}
```

API Gateway – Event Driven

Gösterge Paneli Oluşturma(Dashboard)

Kodu incelersek '**AddCap**' servisi altında '**UseDashboard**' servisini uygulamaya dahil etmekte ve '**/cap-dashboard**' adresi altında erişim sağlanabileceği bildirilmektedir.

API Gateway – Event Driven

CAP Dashboard Published (112) Received (283) Subscribers (2) Servers (0) [Back to site](#)

Succeeded (283)

Failed (0)

Received Messages

Message group Message name Message body [Query](#)

[ReExecution](#) Items per page: 10 20 50 100 500

<input type="checkbox"/>	Id	Version	Group	Name	Retries	Expires At
<input type="checkbox"/>	#1281744715820736512	v1	group1.v1	producer.transaction	0	in a day
<input type="checkbox"/>	#1281744715820736513	v1	group2.v1	producer.transaction	0	in a day
<input type="checkbox"/>	#1281744688381599745	v1	group2.v1	producer.transaction	0	in a day
<input type="checkbox"/>	#1281744688381599744	v1	group1.v1	producer.transaction	0	in a day
<input type="checkbox"/>	#1281744676809515008	v1	group1.v1	producer.transaction	0	in a day

Postman

File Edit View Help

[New](#) [Import](#) [Runner](#) [My Workspace](#) [Invite](#)

GET https://localhost:5001/api/prod... [+](#) [...](#) No Environment

Untitled Request

GET https://localhost:5001/api/producer [Send](#)

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Param

digitalvizyon.net

API Gateway – Event Driven

CAP Avantajları ;

Her event veritabanı seviyesinde tutularak olası hata yahut kesinti durumlarında veri kaybını önlemektedir.

İletişimin tekrar sağlandığı durumlarda mesaj kuyruk sistemine gönderilmemiş ve expire süresi dolmayan mesajları tekrardan göndermektedir.

Böylece kullanılan mesaj kuyruk sistemi, hatalar yahut kesintilerden izole bir şekilde geliştirme yapılmasına olanak sağlamaktadır.

The screenshot shows the SQL Server Enterprise Manager interface. On the left, the Object Explorer displays the database structure for 'DESKTOP-6VE54A5.M...B - cap.Published'. The 'Tables' folder is expanded, showing a list of tables including 'System Tables', 'FileTables', 'External Tables', 'Graph Tables', 'cap.Published', 'cap.Received', and 'dbo._EFMigrationsHistory'. The 'cap.Received' table is selected. In the center, the 'Table List' pane shows a list of tables with columns 'Id', 'Version', and 'Name'. The 'cap.Received' table is highlighted. On the right, the 'Table Content' pane displays the data for the 'cap.Received' table, showing columns 'Name', 'Content', and 'Retries'. The table contains 118 rows of data, with the first row showing 'product.transac...' and '3' in the 'Retries' column.

Docker Container & Redis

Neden Docker Container ?

Chocolatey ile Windows ortamına yapılan Redis kurulumlarında son versiyonunu yüklemekte birtakım problemler yaşamaktayız.

Halbuki Linux ortamda ayağa kaldırılan Docker Containerlar ise Redis sistemlerin en güncel sürümü üzerinde bizlere çalışma imkanı tanımaktadırlar.

Buna ilave olarak windows işletim sistemine Redis serverı kurabilmek için yapılacak birçok ilave iş yükü ve alan tahsisinden bizleri kurtarmakta, tek bir imaj ile Redis sunucusunu ayağa kaldırmamıza imkan tanımaktadır.

Docker Container & Redis

REDIS Container Ayağa Kaldırma

Redis serverı bir containerda ayağa kaldırabilmek için hub.docker.com/_/redis adresindeki imajı kullanacağız.

```
docker run --rm -p 6379:6379 --name rediscontainer -d redis
```


Docker Container & Redis

REDIS Test

Redis sunucuyu container 'da ayağa kaldırdığımızdan dolayı test amaçlı bir tek client 'ı ayağa kaldırmamız yeterli olacaktır.

Bunun için **redis-cli -h localhost -p 1923**

kodunu çalıştırmanız yeterlidir.

Düşen prompt 'ta PING-PONG oluyorsa redis ayakta demektir.

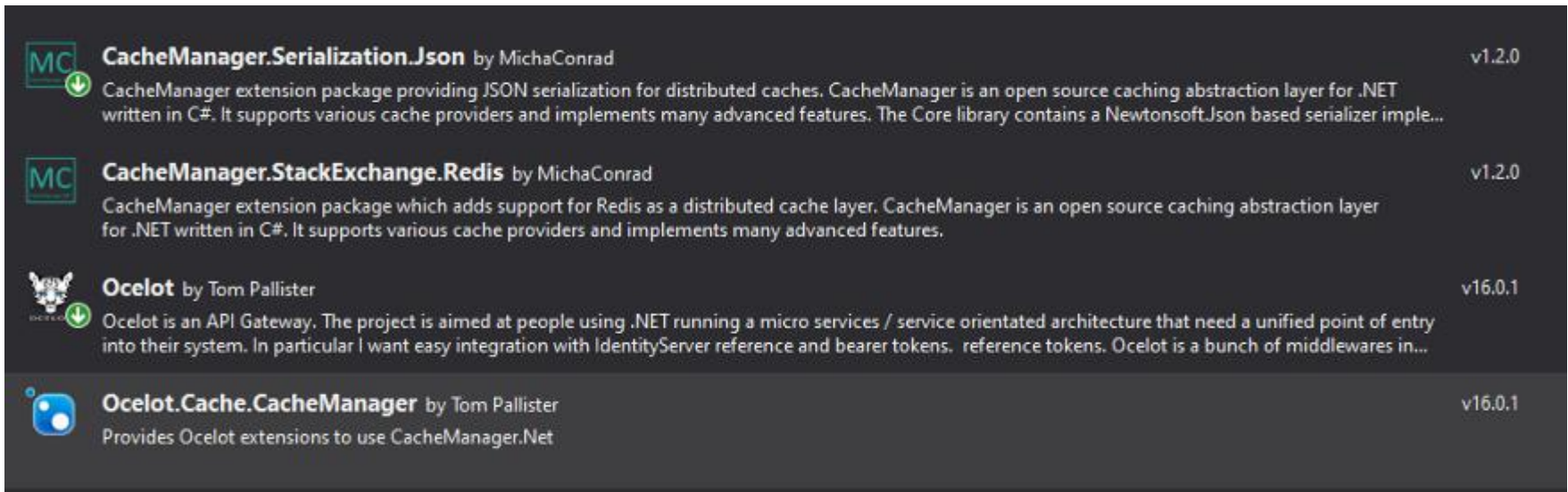
API Gateway - Caching





Asp.NET Core- Microservice uygulamalarında response ' ların *distributed cache* olarak

Redis sunucusunda nasıl saklanacağını inceleyelim.

API Gateway - Caching

Öncelikle gerekli Nuget paketlerini yükleyelim ;



	CacheManager.Serialization.Json by MichaConrad CacheManager extension package providing JSON serialization for distributed caches. CacheManager is an open source caching abstraction layer for .NET written in C#. It supports various cache providers and implements many advanced features. The Core library contains a Newtonsoft.Json based serializer imple...	v1.2.0
	CacheManager.StackExchange.Redis by MichaConrad CacheManager extension package which adds support for Redis as a distributed cache layer. CacheManager is an open source caching abstraction layer for .NET written in C#. It supports various cache providers and implements many advanced features.	v1.2.0
	Ocelot by Tom Pallister Ocelot is an API Gateway. The project is aimed at people using .NET running a micro services / service orientated architecture that need a unified point of entry into their system. In particular I want easy integration with IdentityServer reference and bearer tokens. reference tokens. Ocelot is a bunch of middlewares in...	v16.0.1
	Ocelot.Cache.CacheManager by Tom Pallister Provides Ocelot extensions to use CacheManager.Net	v16.0.1

API Gateway - Caching

Ocelot: Bir API Gateway çözümüdür. İçerisinde routing, caching, rate limiting vb. bir çok özelliği barındırır. Detaylı bilgiye [buradan](#) ulaşabilirsiniz.

Ocelot.Cache.CacheManager: Ocelot paketi içerisinde Caching mekanizmasını kullanmak istersek yüklemek zorunda olduğumuz pakettir. API Gateway'in cachelemeyi hangi provider üzerinde yapmasını istersek bu paket aracılığıyla belirteceğiz.

CacheManager.StackExchange.Redis: Redis yapısını Ocelot üzerinde kullanabilmek için gereken pakettir. Bu paket sonrasında Redis için ayrı bir paket kurmamıza gerek kalmamaktadır.

CacheManager.Serialization.Json: Redis üzerinde verileri JSON serialize ederek tutmamızı sağlayacak olan pakettir.

API Gateway - Caching

Ocelot: Bir API Gateway çözümüdür. İçerisinde routing, caching, load-balancing vb. bir çok özelliği barındırır.

Ocelot.Cache.CacheManager: Ocelot paketi içerisinde Caching mekanizmasını kullanmak istersek yüklemek zorunda olduğumuz pakettir. API Gateway'in cache 'lemeyi hangi provider üzerinde yapmasını istersek bu paket aracılığıyla belirteceğiz.

CacheManager.StackExchange.Redis: Redis yapısını Ocelot üzerinde kullanabilmek için gereken pakettir. Bu paket sonrasında Redis için ayrı bir paket kurmamıza gerek kalmamaktadır.

CacheManager.Serialization.Json: Redis üzerinde verileri JSON serialize ederek tutmamızı sağlayacak olan pakettir.

API Gateway - Caching

API Gateway Yapılandırması

Proje içerisine yeni bir JSON dosyası ekleyelim ve ismini “**ocelot.json**” yapalım.

Bu yapılandırma dosyası Ocelot **API Gateway** çözümünün bel kemiğidir.

API 'ya gelen istekler burada belirtilen parametreler doğrultusunda bir takım **middleware** (ara katman) geçerek *-gerekiyorsa-* projemize ulaşır.

API Gateway - Caching

Ardından dosyamızın ilgili route içeriğini aşağıdaki gibi şekillendirelim:

```
{  
  "DownstreamPathTemplate": "/api/customer",  
  "DownstreamScheme": "https",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 5001  
    }  
  ],  
  "UpstreamPathTemplate": "/api/gateway/customer",  
  "UpstreamHttpMethod": [ "Get" ],  
  "FileCacheOptions": {  
    "Region": "customer",  
    "TtlSeconds": 120  
  }  
}
```

API Gateway - Caching

Routes: Projemiz için belirlediğimiz adreslere kullanıcı istek attığında arka planda hangi API 'ya gideceğini ayarladığımız konfigürasyondur.

Downstream: Aslında burada kastettiğim “downstream” ile başlayan tüm ayarlardır. Bu ayarlar projemizde yer alan API 'lere ait olan yönlendirmelerdir.

Upstream: Upstream ayarları ise kullanıcıların istekte bulunacağı adresleri ve istek tipini belirtmektedir.

FileCacheOptions: Cache servisi konfigürasyonunu burada yapmaktayız. **TtlSeconds** cache süresini, **Region** ise cache anahtarını (key) ayarlamaktadır.

GlobalConfiguration: Son olarak da API Gateway'in ayağa kalkınca kullanacağı ana adresi burada ayarlamaktayız.

API Gateway - Caching

Projemizdeki bu ocelot.json dosyasını açıklamak gerekirse;

<http://localhost:5001/api/customer>

adresine istekte bulunan bir client arkaplanda

<http://localhost:5005/api/gateway/customer>

adresine yönlendirilmektedir. *(Burada mikro servis mimarisi kullanarak farklı host adreslerine de yönlendirme yapılabilirdi)*

Ardından eğer tekrar aynı istekte bulunulursa, API Gateway bu sefer **projeye gitmek yerine**

Redis içerisindeki cache üzerinden veriyi döndürecektir.

API Gateway - Caching

Bu konfigürasyon dosyasını **Program.cs** üzerindeki CreateHostBuilder metodu içerisinde aşağıdaki şekilde kullanarak, ayarları tanımlayalım:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>  
  
Host.CreateDefaultBuilder(args)  
  
.ConfigureAppConfiguration((host, config) => { config.AddJsonFile("ocelot.json"); })  
  
.ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>();  
  
});
```

API Gateway - Caching

Startup.cs dosyamızda ise Ocelot ve Redis caching yapısını aşağıdaki şekilde konfigüre etmemiz gerekmektedir:

API Gateway - Caching

```
services.AddOcelot().AddCacheManager(x =>
{
    x.WithRedisConfiguration("redis",
        config =>
        {
            config.WithAllowAdmin()
                .WithDatabase(0)
                .WithEndpoint("localhost", 6379);
        })
        .WithJsonSerializer()
        .WithRedisCacheHandle("redis");
});
```

API Gateway - Caching

ConfigureServices metodu içerisinde Ocelot servisini ayarlarken, cache yöneticisi olarak **Redis** kullandık. Sonrasında bu konfigürasyona “*redis*” ismini verdik.

Konfigürasyon; Redis sunucusunun **localhost:6379** (varsayılan) adresinde olduğunu, **0** index numaralı veritabanını **admin** yetkileri ile kullanacağımızı belirtmektedir.

Ayrıca verilerimizi **JSON Serialize** ederek saklayacağımızı ve Ocelot üzerinde handle edeceğimiz Redis ayarının “*redis*” ismindeki konfigürasyon olduğunu belirlemiş olduk.

