

RabbitMQ ile Asenkron İletişim

Bir web sitesinde “Siparişi Tamamla” veya “Kaydet” butonuna bastığınızda ve ekran saniyelerce donup kaldığında, o anda dönüp duran yükleme ikonu, aslında sistemin o an çaresizce her işi aynı anda yapmaya çalıştığını gösterir.

Geleneksel yani senkron mimarilerde, kullanıcı tarafından bir istek yollandığında sistem bütün işlemleri sırasıyla tek tek yapmak zorundadır. Veritabanı işlemleri, fatura oluşturma, e-posta API’sine istek atmak, bu zincirin hepsi bitmeden kullanıcıya “İşlem Başarılı” mesajı dönülemez. Ya bu zincirdeki bir servis o an olması gerektiğinden çok yavaş çalışıyorsa? Kullanıcı boşu boşuna beklemiş olmaz mı?

İşte bu tam bu noktada ortaya Message Broker yani Mesaj Kuyruk sistemleri, özellikle de sektörün en popüler kuyruk sistemlerinden biri olan RabbitMQ çıkıyor.

Mesaj Kuyruk Sistemi Nedir?

En basit tabirle, Mesaj Kuyruk Sistemi, uygulamalar arasındaki “Postane”dir. Sistemlerin birbiriyle direkt olarak çalışması yerine aralarında bir aracı kullanılmasını sağlar. Bu da sistemleri sıkı sıkıya bağlı (tightly-coupled) olmasındansa gevşek bağlı (decoupled) olmasını sağlar. Yani bir sistem çöktüğünde diğeri çalışmaya devam edebilir.

Bu mesaj kuyruk sistemleri öncelikle mesajlaşma uygulamalarında kullanılmaya başlandı. Zamanla değişen ihtiyaçlardan dolayı, projelerin ve sistemlerin boyutlarının büyümesi ve kullanıcılardan gelen isteklerin yoğunlaşması, bu gelen isteklere verilen cevapların zamanının uzaması gibi durumlardan dolayı kullanım alanları arttı. Bankaların kullandığı para transfer, EFT vb. İşlemlerde ve yoğun kullanıcı isteği alan e-ticaret sistemlerinin vazgeçilmezi haline geldi.

Örnekleştirmek istersek; Postane sistemi gibi düşünebiliriz.

Süreç olarak nesne kargoya verilir, kargo firması kendine has iş akışlarına göre nesneyi işleme alır ve göndericiye (kurye) ulaştırır, kargo alıcıya ulaşana dek alıcı kendi işlerini yapmaya devam eder kargonun gelmesini beklemeyiz.

Kişisel olarak karşılaştığım ve RabbitMQ kullanmam gereken durumlardan biri de stajımda geliştirdiğim kişisel projemde İBB'nin otopark API'ndan veri çekerken bütün veriyi aynı anda veritabanına kaydetmektense verileri 50'li batchler halinde ayırıp her veri paketini kuyruğa yollayarak verilerin parça parça kaydedilmesini sağladım.

Temel Kavramlar

- Producer (Publisher): Mesajı oluşturup kuyruğa yollayan uygulamadır.
- Consumer: Kuyruğu dinleyip gelen mesajı alıp işleme sokan uygulamadır.
- Queue: Mesajların tutulduğu kuyruk alanıdır. Diskte veya RAM bellekte tutulabilir. Verilerin tutulduğu yere göre avantajlar ve dezavantajları vardır. RAM'de bulunan verilere erişim çok hızlı olur fakat verinin kaybolma riski de çok yüksektir. Diskte bulunan verilerin kaybolma riski çok azdır fakat onlara da erişim daha yavaş olur.
- Routing Key: Mesajın yönlendirileceği anahtardır.
- Exchange: Producer'den mesajı (veriyi) alır ve routing key'e göre mesajları ilgili kuyruğa iletir.

Exchange Tipleri

- 1- Direct Exchange: "Routing key" bilgisi kuyruğa yazılır. Consumer da bu anahtarlara göre bir kuyruğa işlem yapar.
- 2- Fanout Exchange: Mesajlar exchange'de bulunan tüm kuyruklara gönderilir. Yalnızca routing key olanlar göz ardı edilir.
- 3- Topic Exchange: Routing key'e göre farklı kuyruklara yazma işlemleri bu tipte yapılır.
- 4- Headers Exchange: Mesajlar routing key yerine header içerir ve eşleştirmeler bu headerlara göre yapılır.

Her Yerde RabbitMQ Kullanmalı mıyız?

Hayır. Eğer sistemde asenkron çalışması gereken arka plan görevleri yoksa, anında cevap dönmesi gereken basit CRUD işlemlerinden oluşan bir API'dan oluşan bir sistemse RabbitMQ eklemek gereksiz olur. Ekstra maliyet, sunucu yönetimi, karmaşıklık getirir.

Özet

RabbitMQ, günümüzdeki mikroservis sistemlerde ve yoğun trafikli sistemlerde bir lüks değil, gereksinimdir. Doğru senaryoda kullanıldığında kullanıcı deneyimini iyileştirir, sunucuları rahatlatır. “Bu işlem kullanıcıyı bekletmeli mi?” sorusunun cevabı “Hayır” ise, o sistemde RabbitMQ kullanmanız gerekir.

Kod Örnekleri

```
using (var dbConnection = new SqlConnection("Your_Connection_String_Here"))
{
    await dbConnection.ExecuteAsync(
        "sp_CreateReservation",
        new
        {
            RoomId = request.RoomId,
            CustomerName = request.CustomerName,
            Email = request.Email
        },
        commandType: CommandType.StoredProcedure
    );
}
```

Dapper gibi mikro-ORM'ler ile doğrudan Stored Procedure çağırmak veritabanı işlemini milisaniyeler içerisinde tamamlar. Kullanıcı sadece bu işlemi bekler. Eğer fatura veya e-posta gönderme gibi ağır işlemleri de bu blokta yazmış olsaydık kullanıcı o işlemleri de beklemek zorunda kalacaktı.

```
var factory = new ConnectionFactory() { HostName = "localhost" };
await using var connection = await factory.CreateConnectionAsync();
await using var channel = await connection.CreateChannelAsync();

await channel.QueueDeclareAsync(
    queue: "reservation_queue",
    durable: true,
    exclusive: false,
    autoDelete: false,
    arguments: null
);
```

RabbitMQ kütüphanesinin en güncel sürümüyle birlikte artık her şey async/await tabanlı olduğu için sunucu kaynakları bağlantıları beklerken kilitlenmiyor.

Ayrıca “durable : true” parametresi de burada hayat kurtarır. Eğer RabbitMQ sunucusu herhangi bir sebepten dolayı kesinti yaşar ve yeniden başlaması gerekirse, bu parametre sayesinde kuyruktaki mesajlar silinmez.

```

var messagePayload = new { request.CustomerName, request.Email, request.RoomId };
var messageJson = JsonSerializer.Serialize(messagePayload);
var body = Encoding.UTF8.GetBytes(messageJson);

await channel.BasicPublishAsync(
    exchange: "",
    routingKey: "reservation_queue",
    mandatory: false,
    basicProperties: new BasicProperties(),
    body: body
);

```

Peki veriyi neden byte'a çeviriyoruz? Çünkü RabbitMQ C# kodu nedir bilmez. Arka planda sadece 0'lar ve 1'ler üzerinden çalışır. Burada yaptığımız şey aslında bir nevi çevirmenlik. Verimizi RabbitMQ'nun anlayacağı dile çevirip kuyruğa öyle yolluyoruz.

```

var consumer = new AsyncEventingBasicConsumer(channel);

consumer.ReceivedAsync += async (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);

    Console.WriteLine($"[x] Yeni mesaj alındı: {message}");

    await SimulateHeavyProcessAsync();
}

```

Eski senkron sistemleri tek gişeli bir banka şubesi gibi düşünebiliriz. İçerideki memur (Consumer) bir müşterinin işlerini hallederken, o işlem bitene kadar sıradaki herkes beklemek zorundadır. Eğer memur veya sistem yavaşlarsa, tüm kuyruk daha çok beklemek zorunda kalır. RabbitMQ yeni güncellemeleriyle bizi bu çileden kurtardı diyebiliriz. Artık memur karşısındaki müşterinin işlemini hallederken onun tamamlanmasını beklemiyor hemen sıradaki müşteriye geçiyor. Böylece aynı anda birden fazla işlem birbirini bloklamadan arka planda halloluyor.

```

    await channel.BasicAckAsync(deliveryTag: ea.DeliveryTag, multiple: false);
};

await channel.BasicConsumeAsync(
    queue: "reservation_queue",
    autoAck: false,
    consumer: consumer
);

```

“autoAck: true” parametresi, kargocunun paketi apartman kapısına bırakıp sisteme “Teslim Edildi” olarak işaretlemesi gibidir. Eğer o paket kapıdan çalınırsa (sunucu çökerse vs.), geçmiş olsun o kargo (mesaj) kayboldu ve müşterinin bundan haberi bile yok. Ama “autoAck: false” yazdığımız zaman “Islak İmza” protokolünü devreye sokuyoruz. RabbitMQ’ya “E-posta veya mesaj müşteriye sağ salım ulaşp ıslak imza almadan bu mesajı sakın silme” diyoruz.