**Order Management System**

**İŞ AKIŞI ÖZET**

Bu sistem aşağıdaki iş akışını uçtan uca desteklemektedir:

* **ORDER CREATION** → Sipariş Oluşturma
* **INVENTORY SERVICE** → Stok Kontrolü ve Azaltma
* **ORDER COMPLETION** → Sipariş Tamamlama
* **INVOICE CREATION** → Fatura Oluşturma

**SİSTEM MİMARİSİ**

Sistem **Event-Driven Microservices** mimarisi ile geliştirilmiştir:

* **API Gateway** (Port 5004) - Dış dünyaya açık tek giriş noktası
* **Order Service** (Port 5002) - Sipariş yönetimi mikroservisi
* **Inventory Service** (Port 5001) - Stok yönetimi mikroservisi
* **Invoice Service** (Port 5003) - Fatura oluşturma mikroservisi
* **RabbitMQ** - Event bus (mesaj kuyruğu)
* **PostgreSQL** - Her servis için ayrı veritabanı

**KURULUM VE ÇALIŞTIRMA**

**ADIM 1: Docker Ortamını Ayağa Kaldırma**

|  |
| --- |
| *# Proje ana dizininde*  *cd C:\CompanyProjects\OrderManagementSystem\docker*  *# Önceki container'ları durdur docker-compose down*  *# Tüm servisleri ayağa kaldır (PostgreSQL, RabbitMQ)*  *docker-compose up -d* |

**ADIM 2: Database Migration'ları**

|  |
| --- |
| *# Order Service için migration*  *dotnet ef migrations add InitialMigration –project*  *src\Services\OrderManagement.OrderService\OrderManagement.OrderService.csproj --output-dir Data/Migrations*  *# Database'i güncelle dotnet ef database update --project*  *src\Services\OrderManagement.OrderService\OrderManagement.OrderService.csproj --context OrderContext* |

**ADIM 3: Mikroservisleri Ayağa Kaldırma**

|  |
| --- |
| *# API Gateway*  *dotnet run --project OrderManagement.API*  *# Order Service dotnet run --project OrderManagement.OrderService*  *# Inventory Service dotnet run --project OrderManagement.InventoryService*  *# Invoice Service dotnet run --project OrderManagement.InvoiceService* |

**ADIM 4: Test Verilerini Kontrol Etme**

|  |
| --- |
| *SELECT \* FROM public."Products"*  *SELECT \* FROM public."StockMovements"* |

**SİPARİŞ OLUŞTURMA SÜRECİ**

**1. HTTP İsteği**

**Endpoint:** POST <http://localhost:5004/api/Orders>

**Headers:**

|  |
| --- |
| *Content-Type: application/json*  *Idempotency-Key: order-test-123456* |

**Request Body:**

|  |
| --- |
| *{*  *"customerId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",*  *"items": [*  *{*  *"productId": "00e46267-6fd8-40ca-989d-802acbe9b187",*  *"productName": "Headphones",*  *"quantity": 1,*  *"unitPrice": 199.99 },*  *{*  *]*  *}* |

**2. İdempotency Mekanizması**

Idempotency-Key header'ı zorunludur. Bu anahtar ile:

* Aynı key ile gelen istekler aynı sipariş olarak kabul edilir
* Tekrar sipariş oluşturulmaz, mevcut sipariş döndürülür
* Sistemde duplicate (çoklama) engellenmiş olur

**3. Detaylı İşlem Akışı**

**HTTP Katmanı (Senkron)**

1. **Client** → API Gateway'e istek gönderir
2. **API Gateway** → Order Service'e HTTP isteği yönlendirir (http://localhost:5002/api/Orders)
3. **Order Service** → İdempotency kontrolü yapar
   * Aynı key ile sipariş varsa: Mevcut sipariş döndürülür
   * Yoksa: Yeni sipariş oluşturulur

**Database İşlemleri**

1. **Order Service** → Orders tablosuna sipariş kaydı atar
2. **Order Service** → OrderItems tablosuna sipariş kalemleri kaydı atar

**Event-Driven Katman (Asenkron)**

1. **Order Service** → OrderCreated eventini RabbitMQ'ya publish eder

**Inventory Service İşlemleri**

1. **Inventory Service** → OrderCreated eventini yakalar
2. **Inventory Service** → Stok kontrolü yapar (CheckStockAvailabilityAsync)
3. **Inventory Service** → Stok miktarlarını düşer (ReduceStockAsync)
4. **Inventory Service** → Products tablosunu günceller
5. **Inventory Service** → StockMovements tablosuna hareket kaydı atar
6. **Inventory Service** → Sonuca göre event publish eder:
   * Başarılı ise: StockReduced eventi
   * Başarısız ise: StockFailed eventi

**Order Completion İşlemleri**

1. **Order Service** → Gelen eventi yakalar
2. **Order Service** → Sipariş durumunu günceller:
   * StockReduced gelirse: Status = Completed
   * StockFailed gelirse: Status = Failed
3. **Order Service** → Sadece başarılı siparişler için OrderCompleted eventi publish eder

**Invoice Service İşlemleri**

1. **Invoice Service** → OrderCompleted eventini yakalar
2. **Invoice Service** → İdempotency kontrolü (aynı sipariş için fatura var mı?)
3. **Invoice Service** → Invoices tablosuna fatura kaydı atar
4. **Invoice Service** → InvoiceItems tablosuna fatura kalemleri kaydı atar
5. **Invoice Service** → External service'e HTTP çağrısı yapar (Mock service)
6. **Invoice Service** → Sonuca göre event publish eder:
   * Başarılı ise: InvoiceCreated eventi
   * Başarısız ise: InvoiceFailed eventi
7. **EVENT AKIŞI**

|  |
| --- |
| *OrderCreated → Inventory Service*  *StockReduced/StockFailed → Order Service*  *OrderCompleted → Invoice Service*  *InvoiceCreated/InvoiceFailed → (Dead Event - Consumer yok)* |

**TEKNİK ÖZELLİKLER**

**İdempotent Endpoints**

* Sipariş oluşturma endpoint'i idempotent'tir
* Aynı Idempotency-Key ile yapılan istekler aynı sonucu üretir
* Tekrar işlem yapılmaz, mevcut kayıt döndürülür

**Event-Driven İletişim**

* Mikroservisler arası iletişim MassTransit + RabbitMQ ile sağlanır
* Asenkron mesajlaşma ile loosely coupled (gevşek bağlı) mimari
* Her servis bağımsız çalışabilir ve ölçeklenebilir

**Database Per Service Pattern**

* Her mikroservisin kendi veritabanı vardır
* Data ownership principle'ı uygulanır
* Eventual consistency sağlanır

**Resilience (Dayanıklılık)**

* Event'ler RabbitMQ kuyruklarında saklanır
* Servis çökse bile mesajlar kaybolmaz
* Retry mekanizmaları mevcuttur

**MONİTORİNG VE DEBUG**

**RabbitMQ Management UI**

* **URL:** http://localhost:15672
* **Kullanıcı:** guest / guest
* Exchange'leri ve queue'ları izleyebilirsiniz
* Mesaj akışlarını gerçek zamanlı takip edebilirsiniz

**API Dokümantasyonu**

* **Gateway Swagger:** http://localhost:5004/swagger
* **Order Service:** http://localhost:5002/swagger
* **Inventory Service:** http://localhost:5001/swagger
* **Invoice Service:** http://localhost:5003/swagger

**Database Bağlantıları**

Her servisin kendi PostgreSQL veritabanı vardır. Connection string'ler appsettings.json dosyalarında tanımlıdır.

**HATA DURUMU YÖNETİMİ**

**Stok Yetersizliği**

* Inventory Service stok kontrol eder
* Yetersiz stok durumunda StockFailed eventi publish edilir
* Sipariş durumu Failed olur
* Fatura oluşturulmaz

**External Service Hatası**

* Invoice Service external API çağrısı yapar
* %90 başarı oranında mock response döner
* Hata durumunda InvoiceFailed eventi publish edilir

**İdempotency Koruması**

* Aynı Idempotency-Key ile gelen istekler işlenmez
* Mevcut sipariş response olarak döndürülür
* Database'de duplicate kayıt oluşmaz

**GELİŞTİRME NOTLARI**

**Dead Events**

* InvoiceCreated ve InvoiceFailed eventleri şu anda consumer'ı yoktur
* Gelecekte notification, accounting gibi servisler eklenebilir

**Scalability**

* Her mikroservis bağımsız deploy edilebilir
* Load balancer ile horizontal scaling yapılabilir
* Database sharding uygulanabilir

**İletişim Modeli Tercihi**

Event-driven mimari tercih edilme sebepleri:

* **Loose Coupling:** Servisler birbirine sıkı bağlı değil
* **Scalability:** Her servis bağımsız ölçeklenebilir
* **Resilience:** Bir servis çökse diğerleri çalışmaya devam eder
* **Eventual Consistency:** Sistemin sonunda tutarlı hale gelmesi yeterli