TEKNOLOJİ POSTASI

Sıfırdan Apache Kafka

Spring Boot ile Gerçekçi Bir Sipariş İşleme Sistemi

Yusuf Ekrem ÜNLÜ
IBM Business Applications Software Developer
Bilgi Birikim Sistemleri

Kafka'yı öğrenmeye başladığımda, çoğu kaynağın ya çok teorik (topic, partition, offset...) ya da ilk satırda koda dalıp "bu neden var?" sorusunu atladığını fark ettim. Bu yazıda farklı bir yol izleyeceğiz: önce neden Kafka'ya ihtiyaç duyduğumuzu hissedeceğiz, sonra adım adım çalışan bir sistem kuracağız.

Yazının sonunda elinde şunlar olacak:

- Docker üzerinde çalışan bir Kafka
- Spring Boot ile yazılmış bir sipariş servisi (Producer)
- Aynı siparişi bağımsızca işleyen iki servis (Stok + Bildirim)
- Partition'larla paralel/ölçeklenen bir mimari

Kafka nedir? (Teknik Olmayan Anlatım)

Bir restoran düşün.
Kafka'sız dünya: Garson, aldığı siparişi vermek için bizzat mutfağa koşar, aşçıya söyler; sonra kasaya koşar, ödemeyi söyler; sonra müşteriye dönüp haber verir. Aşçı meşgulse garson **bekler**. Kasa çökerse sipariş **kaybolur**. Yeni bir görev (mesela "hediye paketi servisi") eklemek istersen, garsona yine yeni bir koşturma eklersin. Her şey birbirine **sıkı sıkıya bağlı**.

Kafka'lı dünya: Garson siparişi bir **fişe yazıp panoya asar** — ve işi biter. Kimseyi beklemez. Aşçı, kasa, bildirim görevlisi... herkes panoyu **kendi hızında** okur ve işini yapar. Biri hasta olsa fiş panoda **durur**, geri gelince kaldığı yerden devam eder. Yeni bir servis eklemek? Sadece panoyu okumaya başlasın — garsonun haberi bile olmaz.

İşte Kafka, o panodur. Teknik adıyla bir event streaming platformu. Servisler birbirine doğrudan konuşmaz; olayları (event) ortak bir yere yazar ve oradan okurlar. Bu "gevşek bağlılık" (loose coupling), Kafka'nın tüm büyüsüdür.

Temel Kavramlar (Kısa Sözlük)

Yazı boyunca tekrar eden çekirdek terimler:

Kavram	Anlamı
`Topic`	Olayların yazıldığı isimli kanal — restorandaki "pano".
`Partition`	Bir topic'in paralel işlenebilen alt parçası; ölçeklemenin temeli.
`Offset`	Bir partition içindeki her mesajın sıra numarası.
`Producer`	Topic'e mesaj (event) yazan taraf.
`Consumer`	Topic'ten mesaj okuyan taraf.
`Consumer Group`	Birlikte çalışan consumer'lar; her mesajı grup içinde tek consumer işler.
`Broker`	Mesajları saklayan ve dağıtan Kafka sunucusu.

Ne İnşa Edeceğiz?

Basit ama gerçekçi bir e-ticaret akışı:

```
                          ┌─> Stok Servisi      (stoğu düşürür)
Müşteri -> POST /orders -> Kafka │
          (Sipariş Servisi)     └─> Bildirim Servisi  (e-posta gönderir)
```

Müşteri sipariş verir -> sipariş Kafka'ya yazılır -> **iki ayrı servis** bu siparişi birbirinden habersiz, paralel işler. Bu, mikroservis mimarilerinin temel desenidir.

Ortam Hazırlığı

İhtiyacın olanlar (hepsi ücretsiz):

- **Docker Desktop —** Kafka'yı çalıştırmak için -> docker.com
- **Java 21 (JDK) —** Spring Boot için -> adoptium.net
- **Git —** kodu klonlamak için -> git-scm.com

Maven kurmana gerek yok; proje kendi Maven'ını (wrapper) indiriyor.

Adım 1 — Kafka'yı Docker ile Ayağa Kaldır

Modern Kafka, eskiden zorunlu olan Zookeeper olmadan **KRaft modu** ile çalışıyor. Proje köküne bir docker-compose.yml koyuyoruz (tam hâli repoda):

```yaml
services:
  kafka:
    image: confluentinc/cp-kafka:7.7.1
    ports:
      - "9092:9092"
    environment:
      # İki ayrı "kapı": host için localhost:9092, container'lar için kafka:29092
      KAFKA_ADVERTISED_LISTENERS: EXTERNAL://localhost:9092,INTERNAL://kafka:29092
      # ... (diğer KRaft ayarları repoda)

kafka-ui:
    image: provectuslabs/kafka-ui:latest
    ports:
      - "8080:8080"
    environment:
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:29092
```

Çalıştır:

```
docker compose up -d
```

Docker, Kafka imajını internetten indirip container'ı başlatır.
Birkaç saniye sonra panoyu tarayıcıdan görebilirsin: **http://localhost:8080** (Kafka UI).

Adım 2 - Spring Boot Projesi

start.spring.io üzerinden Java 21, bağımlılıklar olarak **Spring Web, Spring for Apache Kafka** ve **Lombok** seçip projeyi oluşturuyoruz.

application.properties'e Kafka bağlantısını ekliyoruz:

```
spring.kafka.bootstrap-servers=localhost:9092
# Producer: nesneyi JSON'a çevirerek gönder
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
# Consumer: JSON'u tekrar nesneye çevir
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=com.tedlim.orders.model
server.port=8081
```

Adım 3 - Producer (Kafka'ya Yazan Taraf)

Önce siparişi temsil eden basit bir veri modeli (Java record):

```
public record Order(String orderId, String product, int quantity,
                    String customer, double totalPrice) {}
```

Kafka'ya yazan servis:

```
@Service
public class OrderProducer {
    private final KafkaTemplate<String, Order> kafkaTemplate;

    public void sendOrder(Order order) {
        // topic, key (orderId), value (order) — gerisini Spring halleder
        kafkaTemplate.send("orders", order.orderId(), order);
    }
}
```

Ve REST giriş kapısı:

```
@RestController
@RequestMapping("/orders")
public class OrderController {
    private final OrderProducer producer;

    @PostMapping
    public ResponseEntity<String> create(@RequestBody Order order) {
        producer.sendOrder(order);
        return ResponseEntity.ok("Sipariş Kafka'ya gönderildi: " + order.orderId());
    }
}
```

Test edelim:

```
curl -X POST http://localhost:8081/orders \
  -H "Content-Type: application/json" \
  -d '{"orderId":"ORD-1","product":"Fren
Balatasi","quantity":2,"customer":"Ahmet","totalPrice":450.0}'
```

Kafka UI --> Topics --> orders --> Messages: Mesajı görüntüleyebilirsin.

Adım 4 - Consumer (Kafka'dan Okuyan Taraf)

Stok servisi, @KafkaListener ile topic'i sürekli dinler. Yeni mesaj gelince Spring bu metodu **bizim için otomatik** çağırır:

```
@Service
public class StockConsumer {
    @KafkaListener(topics = "orders", groupId = "stock-service")
    public void handleOrder(Order order) {
        // Gelen JSON otomatik olarak Order nesnesine çevrildi
        System.out.println("📦 [STOK] " + order.quantity() + " adet '"
            + order.product() + "' düşüldü (" + order.orderId() + ")");
    }
}
```

Artık POST attığında, sipariş saniyenin onda biri içinde otomatik işleniyor: yaz --> pano --> oku --> işle. Sen sadece REST'e istek attın, gerisi kendiliğinden oldu.

Adım 5 - İkinci Servis: İşte Kafka'nın Asıl Gücü

Bir de Bildirim servisi ekleyelim. **Tek fark: farklı bir groupId.**

```
@Service
public class NotificationConsumer {
    @KafkaListener(topics = "orders", groupId = "notification-service")
    public void handleOrder(Order order) {
        System.out.println("📧 [BILDIRIM] Sayın " + order.customer()
            + ", siparişiniz alındı: " + order.orderId());
    }
}
```

Şimdi **tek bir sipariş** gönderdiğinde hem Stok hem Bildirim servisi onu **bağımsızca** işliyor:

```
PRODUCER  -> ORD-4 Kafka'ya yazıldı
   ↓
STOK      -> 4 adet 'Amortisör' stoktan düşüldü      ┐ aynı mesaj,
BILDIRIM  -> Zeynep Şahin'e e-posta gönderildi       ┘ iki bağımsız servis
```

> 🔑 **Kritik nokta:** Producer kodunda tek satır bile değişmedi. Farklı consumer group'lar, aynı mesajın **birer kopyasını** alır. Yarın "Fatura Servisi" eklemek istesen? Yine producer'a dokunmadan üçüncü bir consumer eklersin.

Adım 6 - Partition ve Ölçekleme

Şimdiye kadar topic'imiz tek parçaydı. Günde milyonlarca sipariş gelseydi tek sıra darboğaz olurdu. Çözüm: topic'i birden çok **partition**'a bölmek.

```
                  ┌─ partition 0:  [msg] [msg] ...
orders topic ─────┼─ partition 1:  [msg] [msg] ...
                  └─ partition 2:  [msg] [msg] ...
```

Topic'i 3 partition yapıp, Stok servisini 3 paralel consumer ile çalıştırıyoruz:

```
@KafkaListener(topics = "orders", groupId = "stock-service", concurrency = "3")
```

Sonuç — iş, partition'lar arasında otomatik bölüşülür:

```
stock-service (3 consumer):
   consumer-1 -> partition 0
   consumer-2 -> partition 1     → 3 partition = 3 paralel işçi
   consumer-3 -> partition 2
```

İki altın kural:

1 Bir partition'ı, bir grup içinde **sadece bir** consumer okur.

2 Mesaj hangi partition'a gider? --> **hash(key) % partition_sayısı**. Aynı anahtar hep aynı partition'a düşer, böylece **o anahtarın sırası korunur.** (Sıra garantisi yalnızca partition içinde geçerlidir, partition'lar arasında değil.)

Yolda Karşılaştığım Tuzak: Advertised Listeners

Kurulumda Kafka UI bir türlü bağlanamadı, sonsuz loading'de kaldı. Sebep şuydu: broker, bağlananlara **localhost:9092** adresini "ilan ediyordu". Bu adres benim bilgisayarımdaki Spring Boot için doğruydu — ama Docker container'ındaki Kafka UI için **localhost**, **kendi container**'ı demekti.

Çözüm — iki ayrı kapı (listener):

EXTERNAL://localhost:9092 --> host'tan bağlananlar (Spring Boot)

INTERNAL://kafka:29092 --> Docker ağı içinden bağlananlar (Kafka UI)

Kural: Kafka'ya kim nereden bağlanıyorsa, broker ona **erişebileceği** bir adres ilan etmelidir.