Event-Driven Architecture ve Kafka

Bu kategorideki her servisler arası çağrının şimdiye kadar senkron olmasının yarattığı zaman ve doğrudan-bilgi bağlantısını Kafka ile kaldırmak: topic/partition kavramları, order-service'in KafkaTemplate ile OrderPlacedEvent yayınlaması, inventory-service'in @KafkaListener ile tepki vermesi, senkron vs asenkron arasında ne zaman hangisinin seçileceği, en az bir kez teslimat garantisinin her tüketiciyi idempotent yapmayı gerektirmesi, ve JSON serialization tercihi.

Orta 26 dk

Son güncelleme: 23 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/event-driven-kafka

---

Event-Driven Architecture ve Kafka

Bu kurstaki her servisler arası çağrı şimdiye kadar senkrondu: order-service inventory-service'i çağırıyor ve bir cevap için BEKLİYOR -- ister sabit kodlanmış bir URL'le, ister load-balanced bir RestClient'la, ister bir circuit breaker'la sarılmış olsun ("Servisler Arası İletişim", "Servis Keşfi ve Eureka" ve "Resilience4j" derslerine bakınız). Bu modelin içine gömülü bir şekli var -- çağıran, callee cevap verene kadar bloke olur, ve çağıranın tam olarak hangi servisi çağıracağını bilmesi gerekir. Bu ders temelde farklı bir şekli tanıtıyor: kimin dinlediğini bilmeden ya da umursamadan, zaten olmuş olan gerçekleri duyuran servisler.

Event-Driven Architecture Nedir?

Event-driven (olay güdümlü) bir mimaride, bir servis bir OLAY YAYINLAR -- zaten olmuş bir şeyin kaydı ("bir sipariş verildi") -- herhangi belirli bir başka servise adreslemeden, bir mesaj broker'ına. Herhangi sayıda BAŞKA servis bu olaya abone olabilir ve kendi zamanlamasında, bağımsız olarak tepki verebilir. Yayıncı hiçbir zaman bir tepki bekleyerek bloke olmaz, ve genellikle hangi servislerin (varsa) dinlediğini bile bilmez.

Neden Var?

Senkron çağrılar ZAMANDA SIKI bir bağlantı yaratır: inventory-service yavaşsa ya da düşükse, order-service'in isteği de yavaşlar ya da başarısız olur -- envanteri güncellemek sipariş veren çağıran için aslında acil olmasa bile. Senkron çağrılar ayrıca servisleri BİRBİRLERİNİ doğrudan bilmeye bağlar -- order-service'in inventory-service'in var olduğunu ve ona nasıl ulaşacağını bilmesi gerekir. Olaylar (event'ler) her iki bağlantıyı da kaldırır:

order-service "bir sipariş verildi"yi yayınlar ve hemen devam eder; buna TEK bir servis tepki versin, ÜÇ tanesi versin, ya da altı ay sonra yeni bir tanesi eklensin, order-service'in kendi kodu hiç değişmez.

Tarihçe

Kafka, LinkedIn'in ürettiği devasa hacimdeki aktivite verisini (tıklamalar, görüntülemeler, mesajlar) işlemek için 2011 civarında LinkedIn'de inşa edildi, ve kısa süre sonra Apache Software Foundation üzerinden açık kaynak yapıldı. Geleneksel bir mesaj kuyruğunun (tipik olarak bir mesaj tüketildiğinde silinir) aksine, Kafka kalıcı, yalnızca-ekleme (append-only) bir LOG etrafında inşa edilmiş -- mesajlar, kaç tüketici tarafından okunduğundan bağımsız olarak yapılandırılmış bir saklama süresi boyunca kalır -- bu, birden fazla, bağımsız servisin AYNI olay akışını, aynı mesajlar için rekabet etmeden tüketebilmesini sağlayan şey. Spring for Apache Kafka (`spring-kafka`), Kafka'nın kendi client kütüphanesini tanıdık Spring Boot konvansiyonlarıyla saran Spring projesi -- `@KafkaListener`, `@GetMapping`'e benzer bir rol oynuyor ("Bir Olayı Tüketmek: inventory-service Tepki Veriyor" bölümüne bakınız).

Kafka'yı (Broker) ve Topic'leri Kurmak

eureka-server, api-gateway ya da config-server'ın aksine, Kafka bu kursta inşa edilen bir Spring Boot uygulaması DEĞİL -- ayrı bir altyapı, zaten çalıştığı varsayılıyor (yerel geliştirme için tek bir Kafka broker'ı). Bir TOPIC, üreticilerin yayınladığı ve tüketicilerin abone olduğu adlandırılmış bir olay kategorisidir (bu derste `order-events`); bir topic ayrıca PARTITION'lara bölünür, ve Kafka sırayı yalnızca TEK bir partition İÇİNDE garanti eder, topic'in tamamında değil.

```
# Added to order-service's existing application.yml -- on top of everything from
# earlier lessons in this category. This is what makes order-service able to
# PRODUCE (publish) events -- nothing here changes what order-service already
# does synchronously (its REST API, its Eureka registration, its Resilience4j-
# wrapped call to inventory-service all stay exactly as they were).

spring:
  kafka:
    bootstrap-servers: localhost:9092       # where the Kafka broker itself
                                             # listens -- Kafka is NOT a Spring
                                             # Boot application like eureka-server
                                             # or config-server, it's a separate
                                             # piece of infrastructure this course
                                             # assumes is already running (see
                                             # "Setting Up Kafka (Broker) and
                                             # Topics")
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
                                             # OrderPlacedEvent is serialized to
                                             # JSON on the way out -- see
                                             # "Serialization: Why JSON Over the
                                             # Wire"
```

> `spring.kafka.bootstrap-servers`, order-service'in Kafka'yı bulmak için ihtiyaç duyduğu TEK bilgi -- burada bunun sabit bir adres olduğuna, Eureka üzerinden çözülmediğine dikkat edin ("Servis Keşfi ve Eureka" dersine bakınız) -- Kafka client'larının kendi broker-keşfi protokolü zaten var, bu yüzden broker'ın kendisi için servis keşfine, order-service'in inventory-service'i doğrudan çağırmasındaki gibi ihtiyaç yok.

Bir Olay Yayınlamak: order-service OrderPlaced'i Yayınlıyor

`OrderPlacedEvent`, bilinçli, ayrı bir sözleşme tipi -- `StockCheckResponse`'un içsel bir modeli doğrudan yeniden kullanmak yerine ayrı olmasının arkasındaki AYNI gerekçe ("Servisler Arası İletişim" dersinin "Kendi Sözleşmen: StockCheckResponse Neden InventoryItem Değil?" bölümüne bakınız) burada da geçerli, yalnızca TERS yönde: order-service'in dış dünyaya söylediği bir gerçek, order-service'in aldığı bir cevap değil.

```
// The EVENT itself -- a fact about something that already happened ("an order WAS
// placed"), not a request asking another service to do something. This is the key
// difference from every synchronous call in this course so far (see the Inter-Service
// Communication lesson's StockClient): a REST call SAYS "check this for me, now, and
// tell me the answer"; an event SAYS "this happened, react to it if you care to, on
// your own time." order-service knows NOTHING about who (if anyone) is listening.
//
// Fields mirror order-service's own Order record (see the Spring Boot Microservice
// Basics lesson's Order.java) -- but this is a DELIBERATE, separate type, not the
// domain record itself: an event is a public CONTRACT other services depend on,
// while Order is order-service's own internal model, free to change independently
// (the same "your own contract" reasoning as StockCheckResponse, see the Inter-
// Service Communication lesson's "Your Own Contract: Why StockCheckResponse Instead
// of InventoryItem?" section).
record OrderPlacedEvent(String orderId, String productName, int quantity) {
}
```

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

// order-service's ONLY new piece for publishing events -- OrderService.create(...)
// (see the Spring Boot Microservice Basics lesson) would call
// publish(order) right after saving a new Order, alongside (NOT instead of) its
// existing synchronous checkStock call to inventory-service (see "Synchronous vs.
// Asynchronous: When to Use Which" for why BOTH can coexist for the same order).
//
// KafkaTemplate is spring-kafka's equivalent of RestClient for synchronous calls --
// a thin, autoconfigured wrapper around the underlying Kafka producer client.
@Component
class OrderEventPublisher {

    private static final String TOPIC = "order-events";

    private final KafkaTemplate<String, OrderPlacedEvent> kafkaTemplate;

    OrderEventPublisher(KafkaTemplate<String, OrderPlacedEvent> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    void publishOrderPlaced(String orderId, String productName, int quantity) {
        OrderPlacedEvent event = new OrderPlacedEvent(orderId, productName, quantity);
        // The KEY (orderId here) determines which Kafka PARTITION a message lands
        // in -- messages with the SAME key always go to the SAME partition, and
        // Kafka guarantees ORDER only within a single partition. Keying by orderId
        // means every event about the SAME order is processed in order, even
        // though DIFFERENT orders may be processed out of order relative to each
        // other (a deliberate, common tradeoff -- see "Setting Up Kafka (Broker)
        // and Topics").
        kafkaTemplate.send(TOPIC, orderId, event);
        // send(...) returns a CompletableFuture and does NOT block waiting for
        // Kafka to confirm -- this call returns almost immediately, unlike every
        // synchronous RestClient call in this course so far.
    }
}
```

`OrderService.create(...)` ("Mikroservis Yapılandırma" dersinin "Domain Modeli: Bu Serviste "Sipariş" Ne Demek?" bölümüne bakınız), yeni bir siparişi kaydettikten hemen sonra `publishOrderPlaced(...)`'ı çağırır -- order-service'in zaten senkron olarak yaptığı her şeyin YANINDA, YERİNE değil.

Bir Olayı Tüketmek: inventory-service Tepki Veriyor

`@KafkaListener`, spring-kafka'nın `@GetMapping`'e karşılığı -- bu consumer group için topic'te yeni bir mesaj geldiğinde Spring, işaretlenmiş metodu otomatik olarak çağırır.

```
# Added to inventory-service's own application.yml -- this is what makes
# inventory-service able to CONSUME events, entirely separate from (and running
# alongside) its existing synchronous REST API that StockClient/ResilientStockClient
# call directly.

spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: inventory-service             # ALL instances of inventory-service
                                                # sharing this group id split the
                                                # topic's partitions between them --
                                                # each event is delivered to only
                                                # ONE instance in the group, never
                                                # to all of them at once (this is
                                                # what lets inventory-service scale
                                                # horizontally for event processing,
                                                # exactly like it already does for
                                                # its REST API)
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: "*"       # tells the JSON deserializer which
                                                  # packages it's allowed to
                                                  # instantiate -- "*" is fine for
                                                  # this course's single-package
                                                  # examples, a real deployment
                                                  # would list specific packages
```

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

// inventory-service's reaction to OrderPlacedEvent -- @KafkaListener is spring-
// kafka's equivalent of @GetMapping/@PostMapping for events instead of HTTP
// requests: Spring calls this method automatically whenever a new message arrives
// on "order-events" for this consumer group (see KafkaConsumerConfig.yml).
//
// An in-memory Set stands in for a real "processed order ids" table here, the
// same simplification OrderService.java's in-memory Map makes for orders
// themselves (see the Spring Boot Microservice Basics lesson) -- a real
// deployment would back this with the service's own database.
@Component
class InventoryEventListener {

    // Guards against processing the SAME event twice (see "At-Least-Once
    // Delivery and Idempotency") -- Kafka can redeliver a message inventory-
    // service already handled, most commonly after a restart before an offset
    // was committed.
    private final Set<String> processedOrderIds = ConcurrentHashMap.newKeySet();

    @KafkaListener(topics = "order-events", groupId = "inventory-service")
    void onOrderPlaced(OrderPlacedEvent event) {
        if (!processedOrderIds.add(event.orderId())) {
            // add(...) returns false if this orderId was ALREADY in the set --
            // this is the SAME order being redelivered, not a new one. Reducing
            // stock a SECOND time for it would silently corrupt inventory counts.
            return;
        }

        // In a real deployment, this would decrement a real stock count in
        // inventory-service's own database, using event.productName() and
        // event.quantity() -- kept as a comment here since the point of this
        // lesson is the EVENT FLOW itself, not inventory-service's persistence
        // layer (already covered conceptually in the Microservices Fundamentals
        // lesson's "Database per Service" section).
        // stockRepository.decrease(event.productName(), event.quantity());
    }
}
```

Senkron vs Asenkron: Hangisi Ne Zaman Kullanılır?

Bu ders, "Servisler Arası İletişim" dersindeki senkron çağrıyı DEĞİŞTİRMİYOR -- ikisi de bir arada var oluyor, ve aralarında seçim yapmak sorulan SORUYA bağlı. "Bu ürün şu anda stokta mı, müşteriye hemen bir cevap gösterebileyim mi?" senkron bir çağrı gerektirir -- çağıran, KENDİ çağıranına cevap verebilmeden önce gerçekten bir cevaba ihtiyaç duyar. "Bir sipariş verildi, er ya da geç envanter kayıtlarını güncelle ve önemseyeni bilgilendir" hiç anlık bir cevaba ihtiyaç duymaz -- bir olay doğal olarak uyar, ve çağıran, yavaş olabilecek ya da geçici olarak düşük olabilecek bir servisi beklerken bloke olmaz.

En Az Bir Kez Teslimat ve Idempotency

Kafka (bu dersin varsaydığı yaygın yapılandırmada) EN AZ BİR KEZ (at-least-once) teslimatı garanti eder -- bir tüketici AYNI olayı birden fazla kez görebilir, en sık bir mesajı işlemeyi bitirdiğini onaylamadan ("commit" etmeden) önce bir yeniden başlatmadan sonra. Bu, bir tüketicinin bir olayı işlemesinin IDEMPOTENT olması gerektiği anlamına gelir -- aynı olayı iki kez işlemek, bir kez işlemekle AYNI etkiye sahip olmalı.

> `InventoryEventListener`'ın `processedOrderIds` koruması tam olarak bu idempotency kontrolü -- o olmadan, yeniden teslim edilen bir `OrderPlacedEvent`, zaten işlenmiş bir sipariş için envanteri İKİNCİ kez azaltır, stok sayılarını sessizce bozar. Bu, event-driven sistemlerde gerçek, yaygın bir hata modu, varsayımsal bir kenar durum değil.

Serialization: Neden Tel Üzerinde JSON

`OrderPlacedEvent`, giderken (`JsonSerializer`) JSON'a, gelirken (`JsonDeserializer`) tekrar bir Java nesnesine serialize edilir -- bu kursun REST API'lerinin zaten kullandığı AYNI format, burada da aynı gerekçeyle seçildi: insan tarafından okunabilir, gelecekteki bir tüketicinin hangi dilde yazılmış olursa olsun çalışır, ve çalışan bir sistemde incelemek için ekstra bir araca ihtiyaç duymaz. (Production Kafka dağıtımları genellikle bunun yerine bir schema registry ile birlikte Avro gibi ikili bir format kullanır, bu okunabilirliğin bir kısmını daha küçük mesajlar ve üreticiler/tüketiciler arasında daha katı, zorunlu kılınan sözleşmelerle takas eder -- bu dersin kapsamı dışında.)

Best Practices

- **Olayları, sıralamanın gerçekten neyi önemsediğini belirleyen bir id ile keyle** (burada `orderId`) -- AYNI varlıkla ilgili olaylar aynı partition'a düşer ve sırayla işlenir; FARKLI varlıklarla ilgili olayların buna ihtiyacı yok.
- **Her tüketiciyi idempotent yap**, yalnızca bu dersin `inventoryService`'ini değil -- en az bir kez teslimat Kafka geneli bir garanti, tek bir topic'e özgü bir şey değil.
- **Bir olayın şeklini, bir REST yanıtıyla aynı şekilde, herkese açık bir sözleşme olarak ele al** -- kontrolünde olmayan başka tüketiciler zaten `OrderPlacedEvent`'in tam alanlarına bağımlı olabilir.
- **Anlık bir cevaba ihtiyaç duymayan gerçekler için olaylara, ihtiyaç duyanlar için senkron çağrılara yönel** -- "Senkron vs Asenkron: Hangisi Ne Zaman Kullanılır?" bölümüne bakınız -- hiçbir yaklaşım diğerinin yerini her yerde almaz.

Yaygın Hatalar

- **Idempotent OLMAYAN bir tüketici yazmak.** En az bir kez teslimat, yeniden teslimatın er ya da geç GERÇEKLEŞECEĞİ anlamına gelir -- bunu nadir bir kenar durum

gibi ele almak, tasarlanması gereken bir kesinlik yerine, gerçek veri bozulmasına yol açar (yukarıdaki uyarıya bakınız).

- **Bir olayı yayınlayıp ondan senkron bir çağrının döndürdüğü gibi ANLIK bir cevap beklemek.** Olaylar, yayıncı tarafından fire-and-forget'tir -- gerçekten bir cevaba ihtiyaç varsa, doğru araç bir olay değil, senkron bir çağrıdır.
- **Tüketiciye o kadar çok mantık koymak ki gizli, belgelenmemiş bir bağımlılık haline gelsin.** `OrderPlacedEvent`'e tepki vermek kritik bir iş mantığıysa, o bağımlılık, kimsenin var olduğunu hatırlamadığı bir listener'a gömülmek yerine görünür ve anlaşılır olmalı.
- **Tek bir Kafka topic'inin ve partition'ının sonsuza kadar ölçekleneceğini varsaymak.** Sıralama garantileri yalnızca bir partition içinde geçerli -- bir topic'in, bir consumer group'un işlemi birden fazla örnek arasında GERÇEKTEN paralelleştirebilmesi için yeterli partition'a ihtiyacı var.

Özet, Cheat Sheet ve Terimler Sözlüğü

Event-driven mimari, bir servisin, kimin tepki verdiğini bilmeden ya da onu beklemeden, gerçekleri (olayları) bir mesaj broker'ına yayınlamasına izin verir -- bu, senkron çağrıların yarattığı hem zaman-bağlantısını hem doğrudan-bilgi-bağlantısını kaldırır. Kafka, olayları topic'lere ve partition'lara organize eder, sırayı yalnızca bir partition içinde garanti eder; `KafkaTemplate` yayınlar, `@KafkaListener` tüketir. Kafka'nın en az bir kez teslimatı, her tüketicinin idempotent olması gerektiği anlamına gelir. Olaylar ve senkron çağrılar farklı sorunları çözer ve aynı sistemde bir arada var olur -- hiçbiri diğerinin yerini almaz.

Hızlı referans:

```
record OrderPlacedEvent(String orderId, String productName, int quantity) {}

// Yayınlamak
kafkaTemplate.send("order-events", orderId, event);   // key = orderId, AYNI
                                                        // siparişle ilgili
                                                        // olayları sırayla tutar

// Tüketmek
@KafkaListener(topics = "order-events", groupId = "inventory-service")
void onOrderPlaced(OrderPlacedEvent event) {
    if (!processedOrderIds.add(event.orderId())) return;   // idempotency koruması
    // ...
}
```

Terimler Sözlüğü

Event (Olay) — Zaten olmuş bir şeyin, belirli bir tüketiciye adreslenmeden yayınlanan kaydı.

Topic — Kafka'da üreticilerin yayınladığı, tüketicilerin abone olduğu adlandırılmış bir olay kategorisi.

Partition — Bir topic'in alt bölümü; Kafka mesaj sırasını yalnızca tek bir partition içinde garanti eder.

Consumer Group — Bir topic'in partition'larını işleme işini paylaşan, her olayın yalnızca bir üyesine teslim edildiği bir tüketici örnekleri kümesi.

Idempotency — Aynı olayı birden fazla kez işlemenin, tam olarak bir kez işlemekle aynı etkiye sahip olması özelliği.