Distributed Transactions

Event-Driven Architecture ve Kafka dersinin açık bıraktığı boşluğu kapatmak -- inventory-service stok rezerve edemezse order-service'in zaten commit ettiği siparişe ne olur: Two-Phase Commit'in mikroservislerde neden nadir olduğu, Saga deseni (yerel işlemler + kompansasyon eylemleri), choreography ile StockReservationFailedEvent üzerinden iki yönlü bir saga akışı kurmak, kompansasyonun bir rollback olmadığı, ve Outbox deseninin bir olayın hiç yayınlanmama riskini kapatması.

Orta 26 dk

Son güncelleme: 23 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/distributed-transactions

---

Distributed Transactions

"Event-Driven Architecture ve Kafka" dersi order-service ile inventory-service'e senkron bir çağrı olmadan iletişim kurmanın bir yolunu verdi -- ama dürüst bir boşluk bıraktı: inventory-service bir siparişin ihtiyaç duyduğu stoku GERÇEKTEN rezerve EDEMEZSE ne olur? O cevap geri geldiğinde order-service siparişi zaten kendi veritabanına COMMIT etmiş oluyor. Tek veritabanlı bir uygulama her iki adımı da bir işlemde sarardı ve başarısızlıkta birlikte geri alırdı -- ama order-service ile inventory-service'in her birinin KENDİ veritabanı var ("Microservices Temelleri" dersinin "Database per Service" bölümüne bakınız), ve ikisini de kapsayan tek bir işlem yok. Bu ders, mikroservislerin bu gerçekliği nasıl ele aldığını kapsıyor.

Distributed Transactions Nedir?

Bir distributed transaction (dağıtık işlem), BİRDEN FAZLA bağımsız veritabanı (ya da servis) üzerindeki, hepsinin BAŞARILI olması ya da hepsinin BİRLİKTE geri alınması gereken bir işlemler kümesidir -- order-service'in veritabanına bir sipariş kaydetmek ve inventory-service'in veritabanında stok rezerve etmek, aslında paylaşılan bir işlemi olmayan iki ayrı veritabanı olsa bile, TEK bir mantıksal iş birimi olarak ele alınır.

Neden Var?

"Database per service" (bilinçli bir mikroservis ödünü, bir gözden kaçırma değil), bu kursun Transaction Management dersinin kapsadığı veritabanı seviyesi ACID garantilerinin yalnızca BİR servisin KENDİ veritabanı İÇİNDE geçerli olduğu, hiçbir zaman ikisi arasında olmadığı anlamına gelir. Ama iş operasyonları yine de rutin olarak servisleri kapsıyor -- bir sipariş

vermek gerçekten envanterin mevcut olmasına bağlı. Bu boşluğu görmezden gelmek onu ortadan kaldırmıyor; yalnızca bir servis bunlar için bilinçli olarak tasarlanmadıkça, başarısızlıkların tutarsız ele alınmasına, ya da hiç ele alınmamasına yol açıyor.

Tarihçe

Bu soruna klasik cevap mikroservislerden çok öncesine dayanıyor: Two-Phase Commit (2PC), 1980'lerin dağıtık veritabanı araştırmasından bir protokol, birden fazla veritabanını merkezi bir koordinatör kullanarak tek bir hep-ya-da-hiç sonucuna koordine eder. Çalışır, ama her katılımcının süreç boyunca KİLİTLİ ve bekler durumda olmasını gerektirir -- tam olarak mikroservis mimarilerinin genellikle kaçınmaya çalıştığı türden sıkı bağlantı ve erişilebilirlik maliyeti ("Microservices Temelleri" dersinin "CAP Teoremine Kısa Bir Bakış" bölümüne bakınız). Hector Garcia-Molina ve Kenneth Salem'in 1987'de bir veritabanı makalesinde tanımladığı Saga deseni ("microservices" bir terim olmadan çok önce), sorunu yeniden çerçeveler: tek büyük koordine edilmiş bir işlem yerine, her biri SONRAKİ bir adım başarısız olursa geri alma yolu tanımlı, bir dizi küçük YEREL işlem. Modern mikroservis pratiği, ve bu ders, Saga yaklaşımını izliyor.

Two-Phase Commit: Mikroservisler Bunu Neden Genellikle Kaçınıyor

2PC iki fazda çalışır: her katılımcı önce HAZIRLANIR (kaynaklarını kilitler, commit EDEBİLECEĞİNİ onaylar) ve geri raporlar; yalnızca herkes hemfikir olduktan sonra koordinatör herkese gerçekten COMMIT etmesini söyler. order-service'in veritabanı ile inventory-service'in veritabanı ikisi de 2PC destekleseydi, bu teknik olarak sorunu çözerdi -- ama her katılımcı, prepare fazından son commit'e kadar kilitli kalır, ve koordinatörün kendisi protokolün ortasında çökerse, katılımcılar SÜRESİZ BLOKE kalabilir. Bu, mikroservislerin genellikle inşa edildiği erişilebilirlikle doğrudan çelişir ("Servis Keşfi ve Eureka" dersinin "Eureka'nın CAP Teoremindeki Yeri: AP Sistemi" bölümüne bakınız -- bu kursta başka bir yerde işleyen aynı AP-eğilimli felsefe) -- bu yüzden 2PC, tarihsel olarak "doğru" cevap olmasına rağmen, gerçek mikroservis sistemlerinde pratikte nadirdir.

Saga Deseni: Bir Dizi Yerel İşlem

Bir saga, tek bir distributed transaction'ı, her biri BİR SONRAKİ başlamadan önce KENDİ servisinin veritabanında tamamen commit edilen bir DİZİ yerel işleme böler. SONRAKİ bir adım başarısız olursa, saga paylaşılan bir işlemi geri almaz (öyle bir şey yok) -- KOMPANSASYON (compensating) eylemleri çalıştırır, zaten başarılı olmuş adımların etkilerini açıkça geri alır.

Choreography: Sipariş Verme Bir Saga Olarak

Bu ders CHOREOGRAPHY kullanıyor -- her servis olaylara tepki verir ve kendi bir sonraki hamlesine karar verir, merkezi bir koordinatör olmadan (ORCHESTRATION saga'sı, her adımı yöneten özel bir koordinatör servisiyle, alternatif -- tek bir akış olarak daha görünür, ama başka bir servisi inşa etme ve bakımını yapma maliyetiyle; burada kapsam dışı). Saga'nın iki adımı var: order-service'in yerel işlemi (siparişi vermek, "Mikroservis Yapılandırma" dersinde zaten işlendi), ve inventory-service'in yerel işlemi (stok rezerve etmek), "Event-Driven Architecture ve Kafka" dersindeki olaylarla bağlanmış.

```
// The COMPENSATING event -- published by inventory-service back onto Kafka
// (see the Event-Driven Architecture & Kafka lesson) when it CANNOT honor an
// OrderPlacedEvent, most commonly because there isn't enough stock. This is
// what makes the saga (see "The Saga Pattern: A Sequence of Local Transactions")
// a two-way conversation instead of a one-way announcement -- order-service
// needs to hear back when the OTHER side of its local transaction didn't
// succeed, so it can undo what it already committed.
record StockReservationFailedEvent(String orderId, String productName, String reason) {
}
```

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

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

// inventory-service's saga PARTICIPANT step -- its LOCAL transaction (reserving
// stock in its own database) followed by ONE of two possible outcomes announced
// back onto Kafka. This replaces the InventoryEventListener from the Event-
// Driven Architecture & Kafka lesson, which only ever succeeded silently -- a
// saga needs the FAILURE path to be just as visible as the success path (see
// "Choreography: Order Placement as a Saga").
@Component
class InventoryReservationListener {

    private static final String STOCK_RESERVATION_FAILED_TOPIC = "stock-reservation-failed";

    private final Set<String> processedOrderIds = ConcurrentHashMap.newKeySet();
    private final KafkaTemplate<String, StockReservationFailedEvent> kafkaTemplate;

    InventoryReservationListener(KafkaTemplate<String, StockReservationFailedEvent>
kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    @KafkaListener(topics = "order-events", groupId = "inventory-service")
    void onOrderPlaced(OrderPlacedEvent event) {
        if (!processedOrderIds.add(event.orderId())) {
            return;   // same idempotency guard as the Event-Driven Architecture &
                      // Kafka lesson's InventoryEventListener -- unchanged reasoning
        }
```

```
        // inventory-service's LOCAL transaction: check and reserve stock in ITS
        // OWN database, and ITS OWN database only -- this never reaches across
        // into order-service's database, which is exactly what "distributed"
        // means here (see "What Are Distributed Transactions?").
        boolean reserved = tryReserveStock(event.productName(), event.quantity());

        if (!reserved) {
            // The COMPENSATION trigger -- inventory-service can't undo an order
            // it never created, so instead it tells order-service (the service
            // that CAN undo it) that this step of the saga failed.
            kafkaTemplate.send(STOCK_RESERVATION_FAILED_TOPIC, event.orderId(),
                    new StockReservationFailedEvent(event.orderId(), event.productName(),
"insufficient stock"));
        }
        // If reserved == true, nothing further is published here -- silence IS
        // the success signal in this simple two-step saga. A longer saga (more
        // participants) would typically publish an explicit "stock reserved"
        // event of its own for the next step to react to.
    }

    private boolean tryReserveStock(String productName, int quantity) {
        // Stands in for a real, ATOMIC "check and decrement" against inventory-
        // service's own database (see the Microservices Fundamentals lesson's
        // "Database per Service" section) -- kept as a comment since the point
        // of this lesson is the SAGA FLOW around this call, not inventory-
        // service's persistence layer.
        // return stockRepository.tryReserve(productName, quantity);
        return true;
    }
}
```

Kompansasyon Eylemleri: Zaten Olmuş Bir Şeyi Geri Almak

Bir kompansasyon eylemi bir veritabanı rollback'i DEĞİLDİR -- order-service'in "sipariş ver" işlemi bu çalışmadan önce zaten başarıyla commit edilmiş. Kompansasyon, sistemi orijinal işlemin hiç olmamış gibi davranmak yerine, yeni, düzeltilmiş bir duruma taşıyan AYRI, açık bir yerel işlemdir (siparişi iptal etmek).

```
// A status order-service's Order (see the Spring Boot Microservice Basics
// lesson's Order.java) would need to track once a saga can fail partway
// through -- CONFIRMED once no compensation has arrived after a reasonable
// window, CANCELLED once OrderCancellationListener processes a
// StockReservationFailedEvent for it. Kept as a standalone type here (rather
// than rewriting Order.java itself) to keep this lesson focused on the SAGA
// FLOW, the same scoping choice InventoryReservationListener's tryReserveStock
// makes for inventory-service's own persistence.
enum OrderStatus {
    PLACED,
    CONFIRMED,
    CANCELLED
}
```

Başarısız Bir Rezervasyona Tepki Vermek: Siparişi İptal Etmek

order-service kompansasyon olayını dinler ve buna karşılık kendi yerel işlemini çalıştırır.

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

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// order-service's COMPENSATING step -- the other half of the saga started in
// InventoryReservationListener. This is what makes the whole flow a saga
// rather than just "fire an event and hope": order-service's OWN local
// transaction (placing the order) already committed by the time this runs, so
// "undoing" it doesn't mean a database rollback -- it means a NEW, explicit
// local transaction that moves the order to CANCELLED (see "Compensating
// Actions: Undoing What Already Happened").
@Component
class OrderCancellationListener {

    private static final Logger log = LoggerFactory.getLogger(OrderCancellationListener.
class);

    @KafkaListener(topics = "stock-reservation-failed", groupId = "order-service")
    void onStockReservationFailed(StockReservationFailedEvent event) {
        // A real deployment would load the order, check it's still PLACED (not
        // already CONFIRMED or CANCELLED -- see "Common Mistakes"), and move it
        // to CANCELLED inside order-service's own transaction. Kept as a
        // comment here for the same reason InventoryReservationListener's
        // tryReserveStock is -- this lesson focuses on the saga's SHAPE, not
        // re-deriving order-service's persistence layer.
        // orderService.cancel(event.orderId(), event.reason());
        log.warn("Order {} cancelled: {}", event.orderId(), event.reason());
    }
}
```

> Bir saga adımının kompansasyonu, siparişin muhtemelen zaten İLERLEMİŞ olabileceğini hesaba katmalı -- `OrderCancellationListener`, sipariş başka bir yoldan bir şekilde zaten `CONFIRMED` olarak işaretlenmişken çalışırsa, onu körü körüne iptal etmek başka bir yerde zaten iletilmiş bir kararla çelişebilir. Kompansasyon yapmadan önce siparişin güncel durumunu kontrol etmek ("Yaygın Hatalar"a bakınız) kompansasyonun kendisi kadar önemli.

Outbox Deseni: Bir Olayı Çökmeye Kaybetmemek

Hâlâ bir boşluk var: `OrderEventPublisher` ("Event-Driven Architecture ve Kafka" dersine bakınız), Kafka'ya `OrderService.create(...)`'in siparişi kaydetmesinden AYRI olarak yayın yapar -- order-service bu iki adım arasında çökerse, sipariş var ama onu duyuran olay hiç

yayınlanmamış olur, ve saga'nın tamamı hiç başlamaz. Outbox deseni bunu, olayı SİPARİŞİ kaydeden AYNI işlem İÇİNDE, AYNI yerel veritabanına yazarak kapatır -- ayrı bir süreç sonra yayınlanmamış olayları okur ve kendi zamanlamasında Kafka'ya gönderir.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

// The OUTBOX pattern, illustrated conceptually (see "The Outbox Pattern: Not
// Losing an Event to a Crash") -- a real implementation needs its own database
// table and a JPA repository; this example uses an in-memory Map the same way
// OrderService.java's own persistence is simplified (see the Spring Boot
// Microservice Basics lesson), to keep the focus on the PATTERN'S shape rather
// than JPA plumbing already covered elsewhere in this course.
//
// The problem this solves: OrderEventPublisher (see the Event-Driven
// Architecture & Kafka lesson) publishes to Kafka SEPARATELY from
// OrderService.create(...) saving the order to its database -- if order-
// service crashes between those two steps, the order exists but the event
// publishing it never happened, and nothing else in the system ever finds out
// an order was placed. The outbox pattern closes that gap by writing the EVENT
// to the SAME local database, in the SAME @Transactional method that saves the
// order -- so either both happen or neither does, using a guarantee order-
// service's own database already gives for free (see the Transaction
// Management lesson).
@Component
class OutboxEventPublisher {

    private final Map<String, OrderPlacedEvent> unpublishedEvents = new
ConcurrentHashMap<>();
    private final KafkaTemplate<String, OrderPlacedEvent> kafkaTemplate;

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

    // Called from INSIDE the same @Transactional method that saves the order
    // itself -- writing to "unpublishedEvents" here is really an INSERT into
    // an outbox TABLE in the real pattern, part of the SAME database
    // transaction, not a separate call to Kafka.
    @Transactional
    void saveForPublishing(OrderPlacedEvent event) {
        unpublishedEvents.put(event.orderId(), event);
    }

    // A SEPARATE process, running on its own schedule, that actually talks to
    // Kafka -- decoupled from the request that created the order. If this
    // fails partway through (Kafka is briefly unreachable), the event stays in
    // the outbox and gets retried on the NEXT scheduled run -- nothing is lost,
    // because publishing was never tied to the original request's own success
    // or failure.
    @Scheduled(fixedDelay = 5000)
```

```
    void publishPendingEvents() {
        List<String> orderIds = List.copyOf(unpublishedEvents.keySet());
        for (String orderId : orderIds) {
            OrderPlacedEvent event = unpublishedEvents.get(orderId);
            kafkaTemplate.send("order-events", orderId, event);
            unpublishedEvents.remove(orderId);
        }
    }
}
```

> Bu örnek, outbox'ın kendisini bir in-memory map'e sadeleştiriyor -- `OrderService.java` 'nın kendi persistence'ının yaptığı AYNI kapsam kararı ("Mikroservis Yapılandırma" dersine bakınız) -- gerçek bir outbox, zamanlanmış yayıncının bir repository sorgusuyla okuduğu, gerçek bir veritabanı tablosudur, bu yüzden garantisi siparişin kendisiyle AYNI yerel veritabanı işleminden gelir.

Best Practices

- **Kısa saga'lar (iki ya da üç adım) için choreography'i tercih et, saga bunun ötesine büyüdüğünde orchestration'a geç** -- birden fazla servis işin içine girince, açık bir koordinatörü izlemek olay zincirlerini izlemekten daha kolay hale gelir.
- **Her saga adımını idempotent yap**, Kafka'nın en az bir kez teslimatının zaten yarattığı AYNI gereksinim ("Event-Driven Architecture ve Kafka" dersinin "En Az Bir Kez Teslimat ve Idempotency" bölümüne bakınız) -- bir saga adımı, herhangi başka bir olay handler'ı gibi tekrar denenebilir ya da yeniden teslim edilebilir.
- **Kompansasyon yapmadan önce bir varlığın güncel durumunu kontrol et** -- bir kompansasyon eyleminin her zaman körü körüne uygulanmasının güvenli olduğunu varsayma (yukarıdaki uyarıya bakınız).
- **Yayınlanması gerçekten kaybedilmesine izin verilemeyen her olay için Outbox desenini kullan** -- sipariş verme tam olarak bu türden bir olay, çünkü saga'nın tamamını başlatan şey bu.

Yaygın Hatalar

- **Varsayılan çözüm olarak Two-Phase Commit'e yönelmek.** Mikroservislerin genellikle kaçınmak için benimsendiği sıkı bağlantıyı ve erişilebilirlik maliyetini yeniden getirir -- "Two-Phase Commit: Mikroservisler Bunu Neden Genellikle Kaçınıyor" bölümüne bakınız.
- **Arada başka hiçbir şeyin olamayacağını varsayan bir kompansasyon eylemi yazmak.** Bir kompansasyon gelmeden önce bir sipariş başka bir yoldan iptal edilmiş, gönderilmiş ya da değiştirilmiş olabilir -- her zaman önce güncel durumu kontrol et.

- **Bir olayı Kafka'ya, bağlı olduğu yerel veritabanı yazımından KOPUK bir şekilde yayınlamak.** Outbox deseni olmadan, ikisi arasındaki bir çökme, saga'nın hiç başlamadığı bir durumda sistemi bırakır.
- **Bir saga'yı çağıranın perspektifinden tek bir atomik operasyon gibi ele almak.** Gerçek bir veritabanı işleminin aksine, `OrderService.create(...)` 'in çağıranı, saga'nın SONRAKİ adımları henüz çalışmadan bile anlık bir yanıt alır -- sipariş saniyeler sonra hâlâ iptal edilebilir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Distributed transaction'lar, "database per service"in kaçınılmaz kıldığı, birden fazla servisin bağımsız veritabanlarını kapsar. Two-Phase Commit bunu kilitleme ve merkezi bir koordinatörle çözer, ama mikroservislerin genellikle kaçındığı bir erişilebilirlik maliyetiyle -- Saga deseni bunun yerine operasyonu, başarısızlık için açık kompansasyon eylemleri olan bir dizi yerel işleme böler. Choreography (bu dersin yaklaşımı) her servisin merkezi bir koordinatör olmadan olaylara tepki vermesini sağlar; orchestration bunun yerine özel bir koordinatör kullanır. Outbox deseni, bir yerel veritabanı yazımı ile ona bağlı olayı yayınlamak arasındaki boşluğu, ikisini de aynı yerel işlemde yazarak kapatır.

Hızlı referans:

```
// Adım 1: order-service'in yerel işlemi (Mikroservis Yapılandırma)
Order order = orderService.create(productName, quantity);
outboxEventPublisher.saveForPublishing(new OrderPlacedEvent(order.id(), productName,
quantity));

// Adım 2: inventory-service'in yerel işlemi, olaya tepki veriyor
@KafkaListener(topics = "order-events", groupId = "inventory-service")
void onOrderPlaced(OrderPlacedEvent event) {
    if (!tryReserveStock(event.productName(), event.quantity())) {
        kafkaTemplate.send("stock-reservation-failed", event.orderId(),
                new StockReservationFailedEvent(event.orderId(), event.productName(),
"insufficient stock"));
    }
}

// Kompansasyon: order-service'in yerel işlemi, başarısızlığa tepki veriyor
@KafkaListener(topics = "stock-reservation-failed", groupId = "order-service")
void onStockReservationFailed(StockReservationFailedEvent event) {
    orderService.cancel(event.orderId(), event.reason());
}
```

Terimler Sözlüğü

Distributed Transaction — Birden fazla bağımsız veritabanı üzerinde, hepsinin başarılı olması ya da birlikte geri alınması gereken bir işlemler kümesi.

Two-Phase Commit (2PC) — Her katılımcıyı, hepsini birlikte commit etmeden önce bir prepare fazında kilitleyen bir protokol; erişilebilirlik maliyeti nedeniyle mikroservislerde nadirdir.

Saga — Her biri bir servisin kendi veritabanında olan, başarısızlık için kompansasyon eylemleri tanımlanmış bir dizi yerel işlem.

Kompansasyon Eylemi (Compensating Action) — Bir saga'daki daha önceki bir adım zaten commit edildikten sonra sistemin durumunu düzelten, ayrı, açık bir yerel işlem.

Outbox Deseni — Bir olayı, tanımladığı veriyle AYNI yerel veritabanı işlemine yazmak, böylece bir çökme ikisini birbirinden ayıramaz.