Distributed Transactions

Closing the gap the Event-Driven Architecture & Kafka lesson deliberately left open -- what happens to an order order-service already committed if inventory-service can't reserve stock: why Two-Phase Commit is rare in microservices, the Saga pattern (local transactions plus compensating actions), building a two-way saga flow with choreography and StockReservationFailedEvent, why compensation isn't a rollback, and how the Outbox pattern closes the risk of an event never being published.

Intermediate 26 min

Last updated: October 5, 2026
Read online: https://www.learnforgex.com/en/topics/distributed-transactions

---

Distributed Transactions

The Event-Driven Architecture & Kafka lesson gave order-service and inventory-service a way to communicate without a synchronous call -- but it left an honest gap open: what happens when inventory-service CAN'T actually reserve the stock an order needs? order-service already committed the order to its own database by the time that answer comes back. A single-database application would wrap both steps in one transaction and roll back together on failure -- but order-service and inventory-service each have their OWN database (see the Microservices Fundamentals lesson's "Database per Service" section), and there is no single transaction that spans both. This lesson covers how microservices handle that reality.

What Are Distributed Transactions?

A distributed transaction is a set of operations across MULTIPLE independent databases (or services) that need to either ALL succeed or ALL be undone together -- placing an order in order-service's database and reserving stock in inventory-service's database, treated as one logical unit of work, even though they're two separate databases with no shared transaction.

Why Does It Exist?

"Database per service" (a deliberate microservices tradeoff, not an oversight) means the database-level ACID guarantees this course's Transaction Management lesson covers only ever apply WITHIN one service's own database, never across two. But business operations routinely span services anyway -- placing an order genuinely does depend on inventory

being available. Ignoring this gap doesn't make it go away; it just means failures get handled inconsistently, or not at all, unless a service explicitly designs for them.

History

The classic answer to this problem predates microservices entirely: Two-Phase Commit (2PC), a protocol from 1980s distributed database research, coordinates multiple databases into a single all-or-nothing outcome using a central coordinator. It works, but it requires every participant to be LOCKED and waiting during the whole process -- exactly the kind of tight coupling and availability cost microservices architectures are usually trying to avoid (see the Microservices Fundamentals lesson's "A Quick Look at the CAP Theorem" section). The Saga pattern, described by Hector Garcia-Molina and Kenneth Salem in a 1987 database paper (long before "microservices" was a term), reframes the problem: instead of one big coordinated transaction, a sequence of smaller LOCAL transactions, each with a defined way to undo it if a LATER step fails. Modern microservices practice, and this lesson, follows the Saga approach.

Two-Phase Commit: Why Microservices Usually Avoid It

2PC works in two phases: every participant first PREPARES (locks its resources, confirms it COULD commit) and reports back; only once everyone has agreed does the coordinator tell everyone to actually COMMIT. If order-service's database and inventory-service's database both supported 2PC, this would technically solve the problem -- but every participant stays locked from the prepare phase until the final commit, and if the coordinator itself crashes mid-protocol, participants can be left BLOCKED indefinitely. This directly conflicts with the availability microservices are usually built for (see the Service Discovery & Eureka lesson's "Where Eureka Sits in the CAP Theorem: An AP System" section for the same AP-leaning philosophy at work elsewhere in this course) -- which is why 2PC is rare in practice across real microservices systems, even though it's the historically "correct" answer.

The Saga Pattern: A Sequence of Local Transactions

A saga breaks one distributed transaction into a SEQUENCE of local transactions, each fully committed within its OWN service's database before the next one starts. If a LATER step fails, the saga doesn't roll back a shared transaction (there isn't one) -- it runs COMPENSATING actions, explicitly undoing the effects of the steps that already succeeded.

Choreography: Order Placement as a Saga

This lesson uses CHOREOGRAPHY -- each service reacts to events and decides its own next move, with no central coordinator (an ORCHESTRATION saga, with a dedicated coordinator service directing every step, is the alternative -- more visible as a single flow, at the cost of another service to build and maintain; out of scope here). The saga has two steps: order-

service's local transaction (placing the order, already covered in the Spring Boot Microservice Basics lesson), and inventory-service's local transaction (reserving stock), connected by the events from the Event-Driven Architecture & Kafka lesson.

```
// 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;
    }
}
```

Compensating Actions: Undoing What Already Happened

A compensating action is NOT a database rollback -- order-service's "place order" transaction already committed successfully before any of this runs. Compensation is a SEPARATE, explicit local transaction that moves the system to a new, corrected state (cancelling the order) rather than pretending the original one never happened.

```
// 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
}
```

Reacting to a Failed Reservation: Cancelling the Order

order-service listens for the compensating event and runs its own local transaction in response.

```
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());
    }
}
```

> A saga step's compensation must account for the order having potentially MOVED ON already -- if `OrderCancellationListener` runs after the order was somehow already marked `CONFIRMED` by a different path, blindly cancelling it could contradict a decision already communicated elsewhere. Checking the order's current status before compensating (see "Common Mistakes") matters as much as the compensation itself.

The Outbox Pattern: Not Losing an Event to a Crash

There's still a gap: `OrderEventPublisher` (see the Event-Driven Architecture & Kafka lesson) publishes to Kafka SEPARATELY from `OrderService.create(...)` saving the order -- if order-service crashes between those two steps, the order exists but the event announcing it never got published, and the whole saga never starts. The Outbox pattern closes this by writing the event to the SAME local database, inside the SAME transaction that saves the order -- a separate process then reads unpublished events and sends them to Kafka on its own schedule.

```
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);
        }
```

```
    }
}
```

> This example simplifies the outbox itself to an in-memory map, the same scoping choice `OrderService.java`'s own persistence makes (see the Spring Boot Microservice Basics lesson) -- a real outbox is a genuine database table, read by the scheduled publisher via a repository query, so its guarantee comes from the SAME local database transaction as the order itself.

Best Practices

- **Prefer choreography for short sagas (two or three steps), orchestration once a saga grows past that** -- an explicit coordinator becomes easier to reason about than tracing event chains once more than a couple of services are involved.
- **Make every saga step idempotent**, the same requirement Kafka's at-least-once delivery already created (see the Event-Driven Architecture & Kafka lesson's "At-Least-Once Delivery and Idempotency" section) -- a saga step can be retried or redelivered just like any other event handler.
- **Check an entity's current state before compensating it** -- don't assume a compensating action is always safe to apply blindly (see the warning above).
- **Use the Outbox pattern for any event whose publication genuinely can't be allowed to get lost** -- order placement is exactly this kind of event, since it's what starts the whole saga.

Common Mistakes

- **Reaching for Two-Phase Commit as a default solution.** It reintroduces the tight coupling and availability cost microservices are usually adopted to avoid -- see "Two-Phase Commit: Why Microservices Usually Avoid It".
- **Writing a compensating action that assumes nothing else could have happened in between.** An order could have been cancelled, shipped, or modified by another path before a compensation arrives -- always check current state first.
- **Publishing an event to Kafka in a way that's disconnected from the local database write it depends on.** Without the Outbox pattern, a crash between the two leaves the system in a state where the saga never even starts.
- **Treating a saga as a single atomic operation from the caller's perspective.** Unlike a real database transaction, the caller of `OrderService.create(...)` gets an immediate response before the saga's LATER steps have even run -- the order might still be cancelled seconds later.

Summary, Cheat Sheet, and Glossary

Distributed transactions span multiple services' independent databases, which "database per service" makes unavoidable. Two-Phase Commit solves this with locking and a central coordinator, but at an availability cost microservices usually avoid -- the Saga pattern instead breaks the operation into a sequence of local transactions with explicit compensating actions for failure. Choreography (this lesson's approach) has each service react to events with no central coordinator; orchestration uses a dedicated coordinator instead. The Outbox pattern closes the gap between a local database write and publishing the event that depends on it, by writing both in the same local transaction.

Quick reference:

```
// Step 1: order-service's local transaction (Spring Boot Microservice Basics)
Order order = orderService.create(productName, quantity);
outboxEventPublisher.saveForPublishing(new OrderPlacedEvent(order.id(), productName,
quantity));

// Step 2: inventory-service's local transaction, reacting to the event
@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"));
    }
}

// Compensation: order-service's local transaction, reacting to the failure
@KafkaListener(topics = "stock-reservation-failed", groupId = "order-service")
void onStockReservationFailed(StockReservationFailedEvent event) {
    orderService.cancel(event.orderId(), event.reason());
}
```

Glossary

Distributed Transaction — A set of operations across multiple independent databases that need to succeed or be undone together.

Two-Phase Commit (2PC) — A protocol that locks every participant during a prepare phase before committing all of them together; rare in microservices due to its availability cost.

Saga — A sequence of local transactions, each in one service's own database, with compensating actions defined for failure.

Compensating Action — A separate, explicit local transaction that corrects the system's state after an earlier step in a saga already committed.

Outbox Pattern — Writing an event to the same local database transaction as the data it describes, so a crash can't separate the two.