Event-Driven Architecture & Kafka

Removing the time-coupling and direct-knowledge-coupling created by every inter-service call in this category being synchronous so far, using Kafka: topic/partition concepts, order-service publishing OrderPlacedEvent with KafkaTemplate, inventory-service reacting with @KafkaListener, when to choose synchronous vs. asynchronous, why at-least-once delivery requires every consumer to be idempotent, and the JSON serialization choice.

Intermediate 26 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/event-driven-kafka

---

Event-Driven Architecture & Kafka

Every inter-service call in this course so far has been synchronous: order-service calls inventory-service and WAITS for an answer, whether through a hardcoded URL, a load-balanced RestClient, or one wrapped in a circuit breaker (see the Inter-Service Communication, Service Discovery & Eureka, and Resilience4j lessons). That model has a shape baked into it -- the caller is blocked until the callee responds, and the caller needs to know exactly which service to call. This lesson introduces a fundamentally different shape: services that announce facts about what already happened, without knowing or caring who's listening.

What Is Event-Driven Architecture?

In an event-driven architecture, a service PUBLISHES an event -- a record of something that already happened ("an order was placed") -- to a message broker, without addressing it to any specific other service. Any number of OTHER services can subscribe to that event and react to it independently, on their own schedule. The publisher never blocks waiting for a reaction, and often doesn't even know which services (if any) are listening.

Why Does It Exist?

Synchronous calls create TIGHT coupling in time: if inventory-service is slow or down, order-service's request is slow or fails too, even if updating inventory isn't actually urgent for the caller who placed the order. Synchronous calls also couple services to knowing about EACH OTHER directly -- order-service has to know inventory-service exists and how to reach it.

Events remove both couplings: order-service publishes "an order was placed" and moves on immediately; whether ONE service reacts to it, THREE do, or a new one is added six months from now, order-service's own code never changes.

History

Kafka was built at LinkedIn around 2011 to handle the sheer VOLUME of activity data (clicks, views, messages) LinkedIn generates, and was open-sourced through the Apache Software Foundation shortly after. Unlike a traditional message queue (which typically deletes a message once it's been consumed), Kafka is built around a durable, append-only LOG -- messages stay for a configured retention period regardless of how many consumers have read them, which is what lets multiple, independent services all consume the SAME stream of events without competing for the same messages. Spring for Apache Kafka (`spring-kafka`) is the Spring project that wraps Kafka's own client library with familiar Spring Boot conventions -- `@KafkaListener` playing a role similar to `@GetMapping` (see "Consuming an Event: inventory-service Reacts").

Setting Up Kafka (Broker) and Topics

Unlike eureka-server, api-gateway, or config-server, Kafka is NOT a Spring Boot application built in this course -- it's separate infrastructure, assumed to already be running (a single Kafka broker, for local development). A TOPIC is a named category of events (`order-events` in this lesson) that producers publish to and consumers subscribe to; a topic is further split into PARTITIONS, and Kafka guarantees ordering only WITHIN a single partition, not across the whole topic.

```
# 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` is the ONE piece of information order-service needs to find Kafka -- notice this is a fixed address here, not resolved through Eureka (see the Service Discovery & Eureka lesson) -- Kafka clients have their own broker-discovery protocol built in, so service discovery for the broker itself isn't needed the way it is for order-service calling inventory-service directly.

Producing an Event: order-service Publishes OrderPlaced

`OrderPlacedEvent` is a deliberate, separate contract type -- the same reasoning behind `StockCheckResponse` instead of reusing an internal model directly (see the Inter-Service Communication lesson's "Your Own Contract: Why StockCheckResponse Instead of InventoryItem?" section) applies here too, just in the opposite direction: a fact order-service is telling the outside world, not a response order-service is receiving.

```
// 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(...)` (see the Spring Boot Microservice Basics lesson's "The Domain Model: What Does "Order" Mean in This Service?" section) would call `publishOrderPlaced(...)` right after saving a new order -- alongside, not instead of, anything order-service already does synchronously.

Consuming an Event: inventory-service Reacts

`@KafkaListener` is spring-kafka's equivalent of `@GetMapping` -- Spring calls the annotated method automatically whenever a new message arrives on the topic, for THIS consumer group.

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

Synchronous vs. Asynchronous: When to Use Which

This lesson doesn't REPLACE the synchronous call from the Inter-Service Communication lesson -- both coexist, and choosing between them depends on the QUESTION being asked. "Is this product in stock right now, so I can show the customer an answer immediately?" needs a synchronous call -- the caller genuinely needs an answer before it can respond to ITS OWN caller. "An order was placed, eventually update inventory records and notify whoever cares" doesn't need an immediate answer at all -- an event fits naturally, and the caller isn't blocked waiting on a service that might be slow or temporarily down.

At-Least-Once Delivery and Idempotency

Kafka (in the common configuration this lesson assumes) guarantees AT-LEAST-ONCE delivery -- a consumer might see the SAME event more than once, most often after a restart before it had confirmed ("committed") that it finished processing a message. This means a consumer's handling of an event must be IDEMPOTENT -- processing the same event twice must have the same effect as processing it once.

> `InventoryEventListener`'s `processedOrderIds` guard is exactly this idempotency check -- without it, a redelivered `OrderPlacedEvent` would decrement inventory a SECOND time for an order that was already processed, silently corrupting stock counts. This is a real, common failure mode in event-driven systems, not a hypothetical edge case.

Serialization: Why JSON Over the Wire

`OrderPlacedEvent` is serialized to JSON on the way out (`JsonSerializer`) and back to a Java object on the way in (`JsonDeserializer`) -- the same format this course's REST APIs already use, chosen here for the same reason: it's human-readable, works across any language a future consumer might be written in, and needs no extra tooling to inspect on a running system. (Production Kafka deployments often use a binary format like Avro with a schema registry instead, trading some of that readability for smaller messages and stricter, enforced contracts between producers and consumers -- out of scope for this lesson.)

Best Practices

- **Key events by an id that determines what ordering actually matters for** (`orderId` here) -- events about the SAME entity land in the same partition and are processed in order; events about DIFFERENT entities don't need to be.
- **Make every consumer idempotent**, not just this lesson's `inventoryService` -- at-least-once delivery is a Kafka-wide guarantee, not something specific to one topic.
- **Treat an event's shape as a public contract**, the same as a REST response -- other consumers you don't control may already depend on `OrderPlacedEvent`'s exact fields.
- **Reach for events for facts that don't need an immediate answer, and synchronous calls for questions that do** -- see "Synchronous vs. Asynchronous: When to Use Which" -- neither approach replaces the other everywhere.

Common Mistakes

- **Writing a consumer that isn't idempotent.** At-least-once delivery means redelivery WILL happen eventually -- treating it as a rare edge case, rather than a certainty to design for, leads to real data corruption (see the warning above).

- **Publishing an event and expecting an immediate answer back from it**, the way a synchronous call returns one. Events are fire-and-forget from the publisher's side -- if an answer is genuinely needed, a synchronous call is the right tool, not an event.
- **Putting so much logic in the consumer that it becomes a hidden, undocumented dependency.** If reacting to `OrderPlacedEvent` is critical business logic, that dependency should be visible and understood, not buried in a listener nobody remembers exists.
- **Assuming a single Kafka topic and partition scales indefinitely.** Ordering guarantees only apply within a partition -- a topic needs enough partitions for a consumer group to actually parallelize processing across multiple instances.

Summary, Cheat Sheet, and Glossary

Event-driven architecture lets a service publish facts (events) to a message broker without knowing or blocking on who reacts to them, removing both the time-coupling and the direct-knowledge-coupling that synchronous calls create. Kafka organizes events into topics and partitions, guaranteeing order only within a partition; `KafkaTemplate` publishes, `@KafkaListener` consumes. Kafka's at-least-once delivery means every consumer must be idempotent. Events and synchronous calls solve different problems and coexist in the same system -- neither replaces the other.

Quick reference:

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

// Producing
kafkaTemplate.send("order-events", orderId, event);   // key = orderId, keeps
                                                        // events about the SAME
                                                        // order in order

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

Glossary

Event — A record of something that already happened, published without addressing it to a specific consumer.

Topic — A named category of events in Kafka that producers publish to and consumers subscribe to.

Partition — A subdivision of a topic; Kafka guarantees message order only within a single partition.

Consumer Group — A set of consumer instances that share the work of processing a topic's partitions, each event delivered to only one member.

Idempotency — The property that processing the same event more than once has the same effect as processing it exactly once.