

Complete Kafka Learning Guide

From Theory to Practice: Learn Kafka fundamentals, then build a real event-driven system

Table of Contents

Part 1: Kafka Fundamentals

1. [What is Kafka?](#)
2. [Core Concepts](#)
3. [Cluster Architecture](#)
4. [Topics and Partitions](#)
5. [Producers](#)
6. [Consumers and Consumer Groups](#)
7. [Message Delivery Guarantees](#)
8. [Offsets and Commits](#)

Part 2: Integration with Spring Boot

9. [Project Setup](#)
10. [Configuration](#)
11. [Creating Topics](#)
12. [Building the Producer](#)
13. [Building Consumers](#)
14. [Error Handling and DLQ](#)

Part 3: Code Walkthrough

15. [Project Structure](#)
 16. [Event Design](#)
 17. [Producer Service Explained](#)
 18. [Consumer Service Explained](#)
 19. [Testing the System](#)
-

Part 1: Kafka Fundamentals

1. What is Kafka?

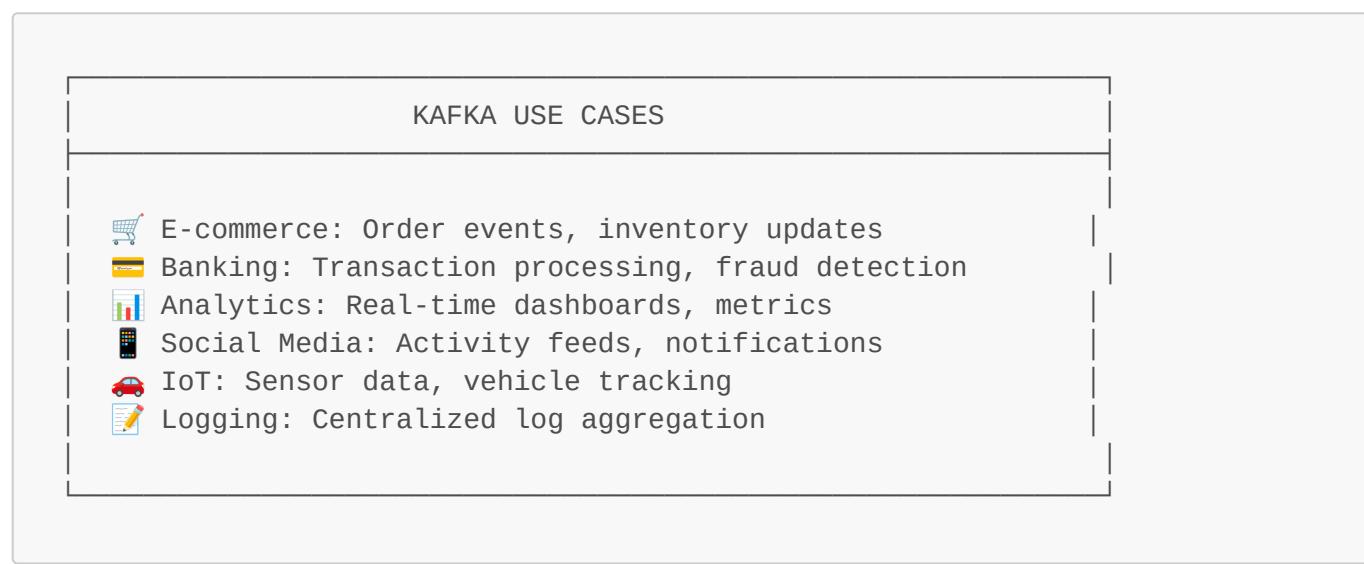
Apache Kafka is a **distributed event streaming platform** used for:

- **Messaging:** Send messages between services (like RabbitMQ)
- **Event Streaming:** Process streams of events in real-time
- **Data Pipeline:** Move data between systems
- **Event Sourcing:** Store events as the source of truth

Why Kafka?

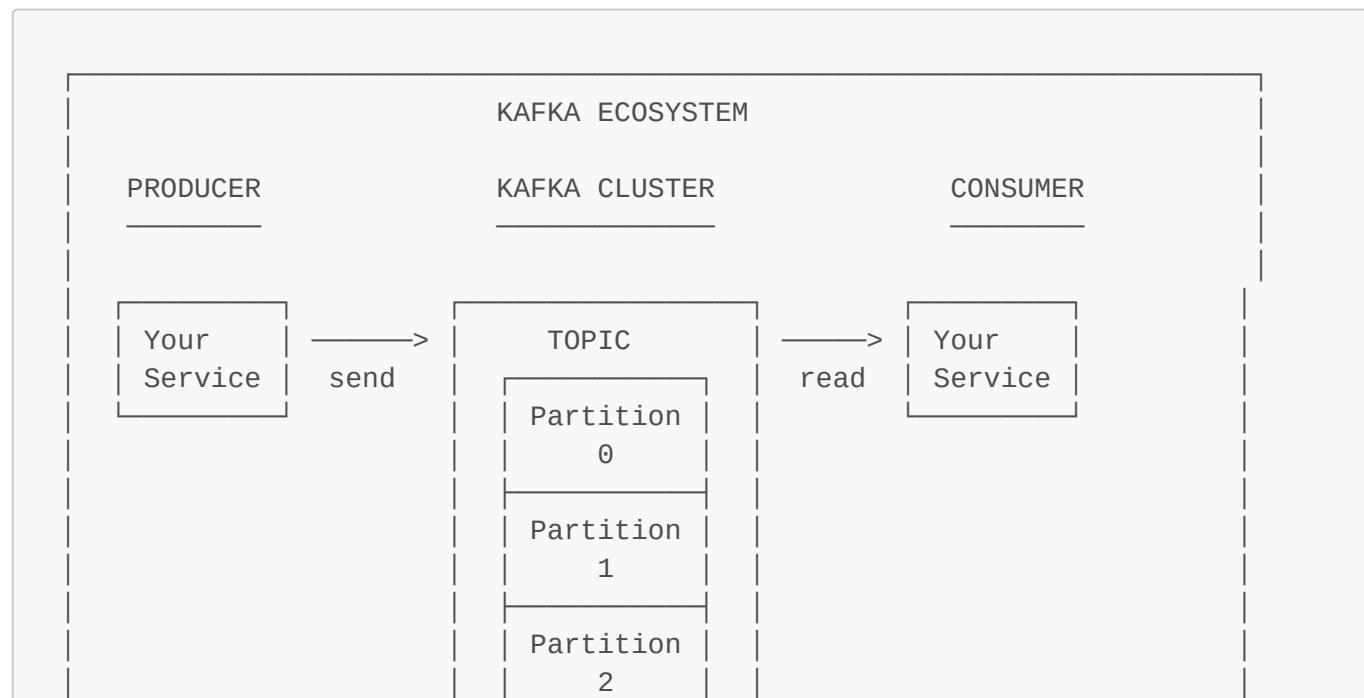
Feature	Kafka	Traditional Queue (RabbitMQ)
Throughput	Millions/sec	Thousands/sec
Storage	Persisted on disk	Memory-based
Replay	Yes (can re-read old messages)	No (message deleted after read)
Scaling	Horizontal (add more brokers)	Limited
Use Case	Event streaming, logs, analytics	Task queues, RPC

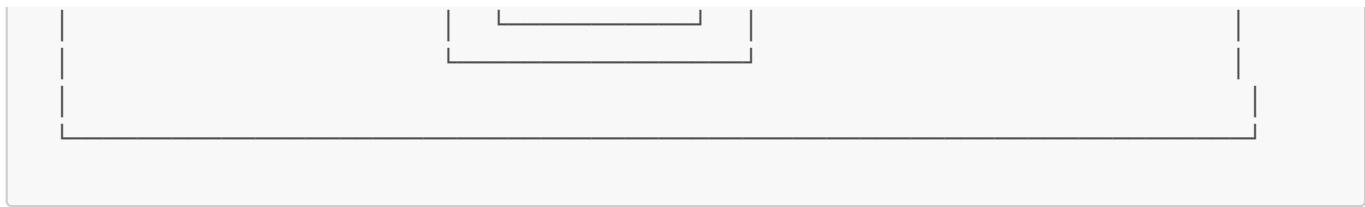
Real-World Use Cases



2. Core Concepts

The Big Picture



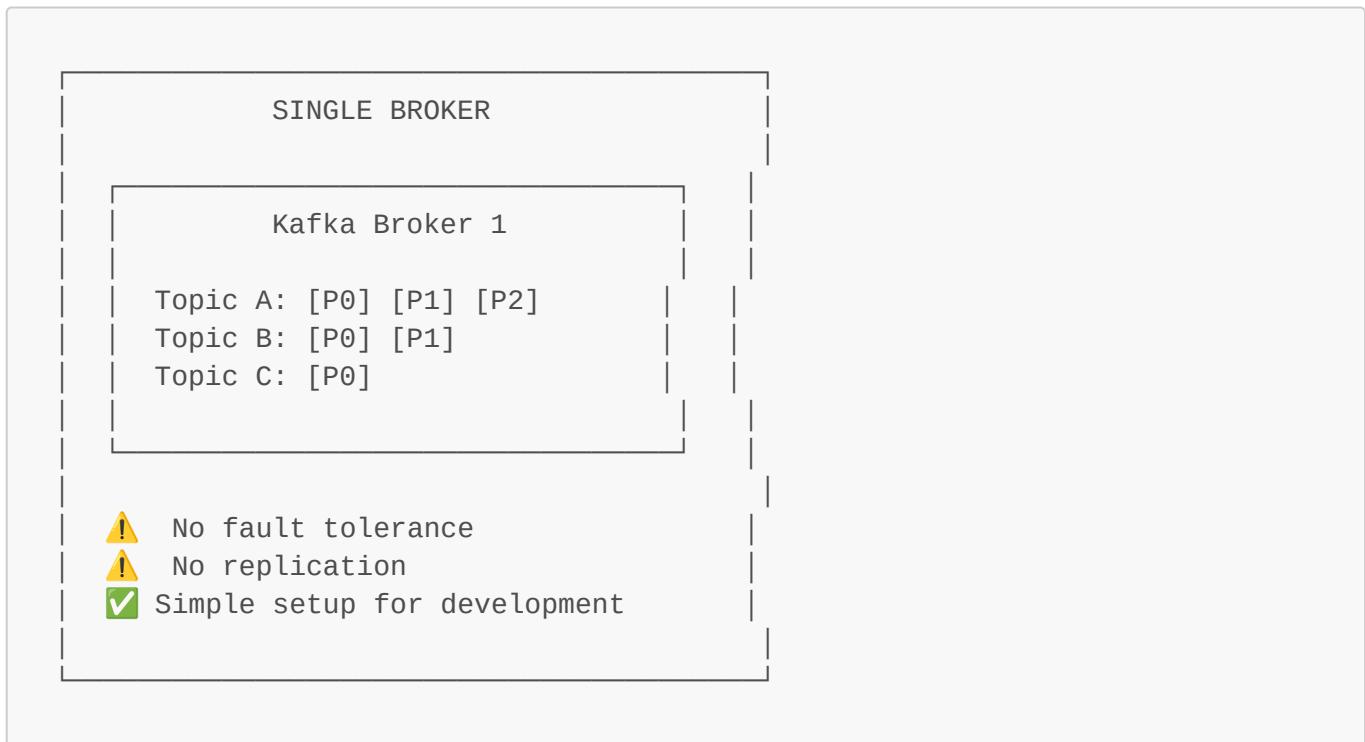


Key Terms

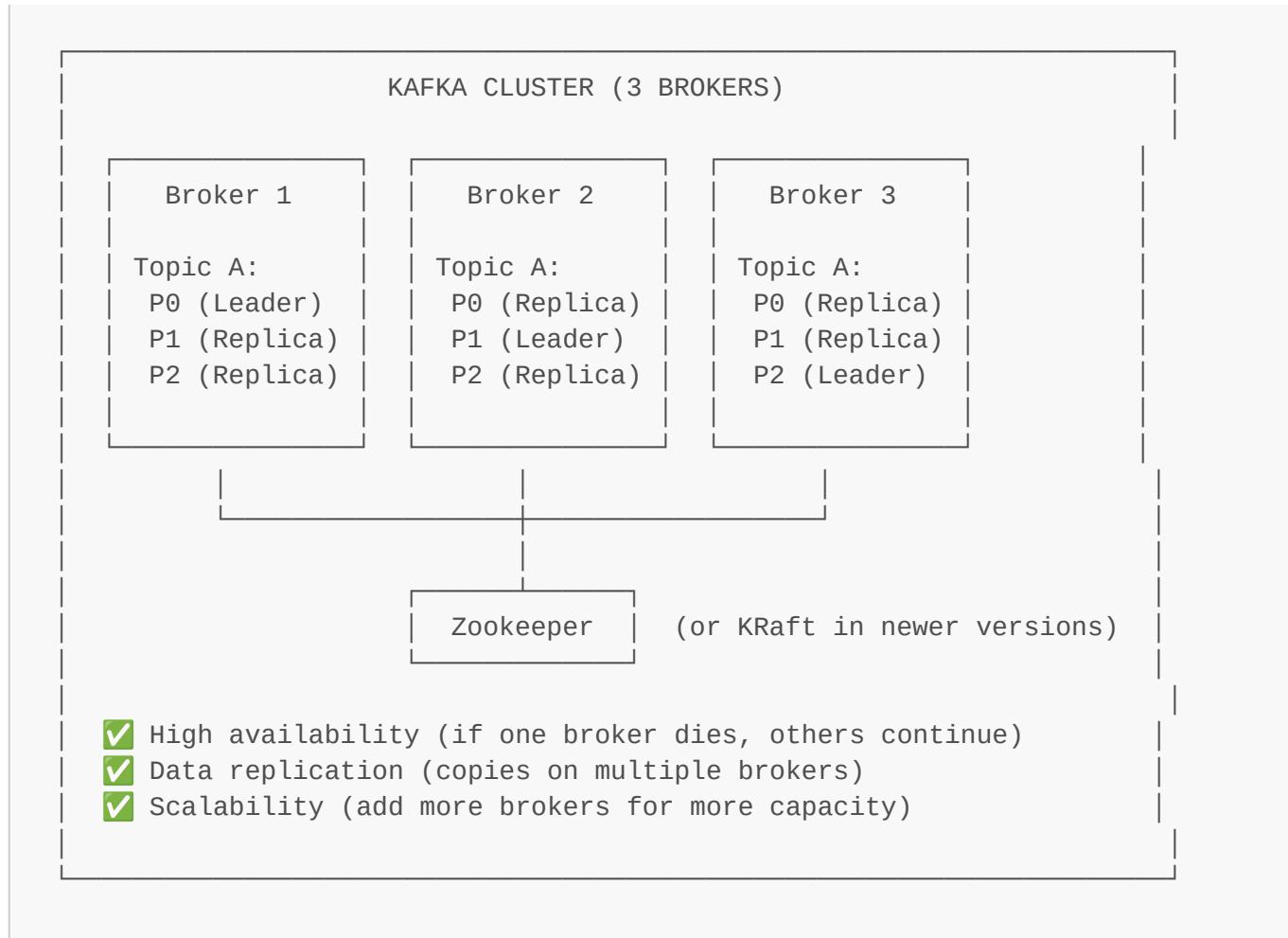
Term	Description	Analogy
Broker	A Kafka server that stores data	A warehouse
Cluster	Multiple brokers working together	Multiple warehouses
Topic	A category/feed name for messages	A folder
Partition	A subset of a topic for parallelism	Subfolders
Producer	Sends messages to topics	The sender
Consumer	Reads messages from topics	The receiver
Consumer Group	Multiple consumers working together	A team of receivers
Offset	Position of a message in partition	Page number in a book
Zookeeper	Manages cluster metadata (legacy)	The manager

3. Cluster Architecture

Single Broker (Development) - Your Current Setup



Multi-Broker Cluster (Production)



Leader and Replica

For each partition:

- ONE Leader: Handles all reads/writes
- MULTIPLE Replicas: Backup copies for fault tolerance

Example: Topic "orders" with 3 partitions, replication factor 3

Broker 1	Broker 2	Broker 3
P0 [LEADER]	P0 [replica]	P0 [replica]
P1 [replica]	P1 [LEADER]	P1 [replica]
P2 [replica]	P2 [replica]	P2 [LEADER]

If Broker 1 dies:

- P0 replica on Broker 2 becomes new Leader
- System continues working!

4. Topics and Partitions

What is a Topic?

A topic is like a **category** or **channel** for messages.

TOPICS

"user-created"	→ All new user registration events
"order-created"	→ All new order events
"payment-completed"	→ All successful payment events
"notification-email"	→ All email notification requests

What is a Partition?

Partitions allow **parallel processing** and **ordering**.

Topic: "order-created" with 3 partitions

Partition 0:	[msg0] [msg3] [msg6] [msg9]	→ Consumer 1
Partition 1:	[msg1] [msg4] [msg7] [msg10]	→ Consumer 2
Partition 2:	[msg2] [msg5] [msg8] [msg11]	→ Consumer 3

- Messages in SAME partition = Ordered
- Messages in DIFFERENT partitions = No order guarantee

How Messages Go to Partitions

MESSAGE → PARTITION ASSIGNMENT

Option 1: No Key (Round-Robin)

Message 1 → Partition 0
Message 2 → Partition 1
Message 3 → Partition 2
Message 4 → Partition 0 (cycles back)

Option 2: With Key (Hash-Based)

Key: "user-123" → hash("user-123") % 3 = Partition 1
Key: "user-456" → hash("user-456") % 3 = Partition 0

- Same key ALWAYS goes to same partition
- Use key when you need ordering for related messages

Example: All orders from user-123 go to same partition

→ Processed in order!

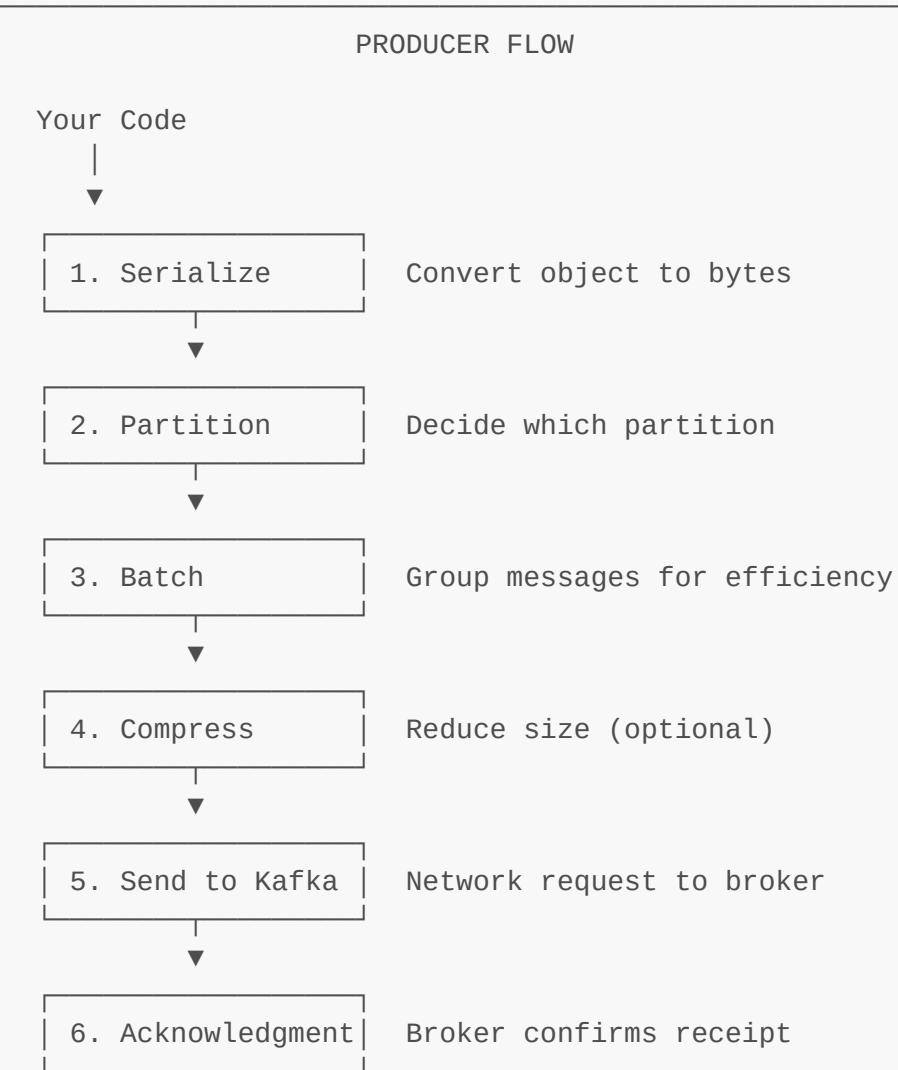
Choosing Partition Count

Partitions	Pros	Cons	When to Use
1	Strict ordering	No parallelism	DLQ, audit logs
3	Good balance	-	Most topics
6+	High throughput	More resources	High-volume topics

Rule of thumb: Partitions = 2 × expected consumer count

5. Producers

How Producers Work



Acknowledgment Modes (acks)

ACKS CONFIGURATION

acks=0 "Fire and Forget"

Producer —send—> Broker
(no wait)

⚡ Fastest

⚠ May lose messages

acks=1 "Leader Acknowledgment" (default)

Producer —send—> Leader Broker —ack—> Producer

✓ Good balance

⚠ May lose if leader dies before replication

acks=all "Full Acknowledgment" (safest)

Producer —send—> Leader —replicate—> All Replicas
└── ack ──> Producer

✓ No data loss

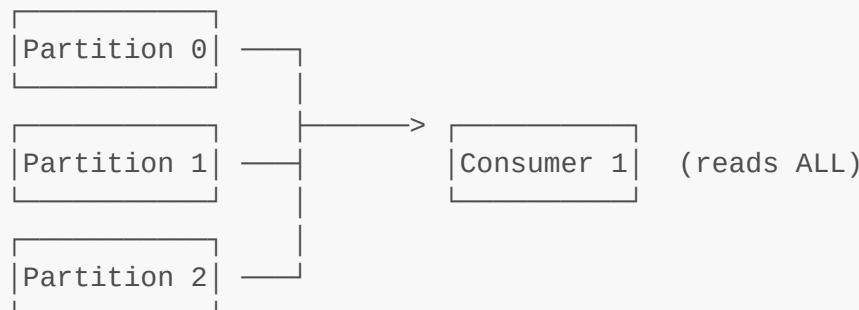
🐢 Slowest

6. Consumers and Consumer Groups

Single Consumer

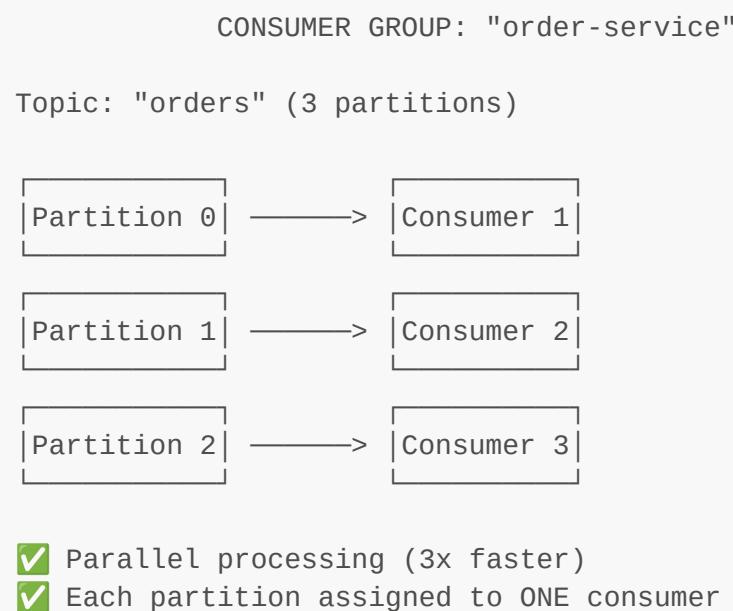
SINGLE CONSUMER

Topic: "orders" (3 partitions)

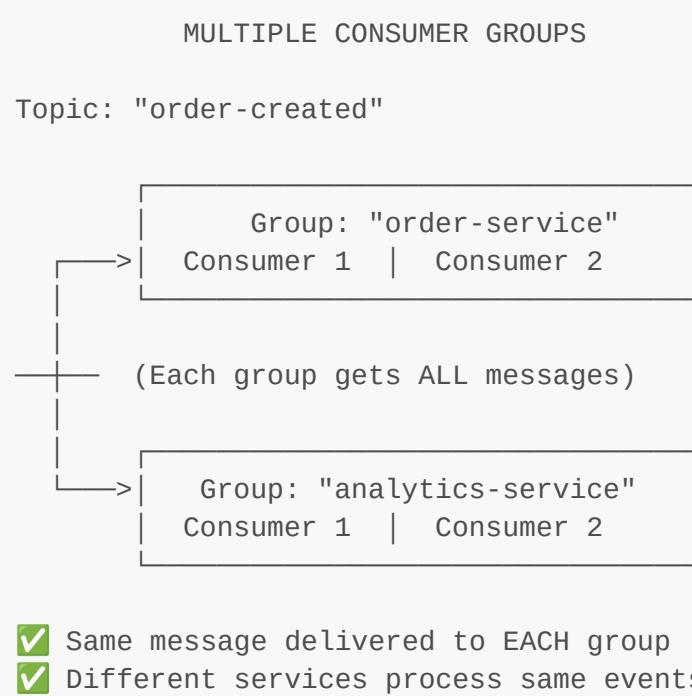


⚠ One consumer = bottleneck

Consumer Group (Parallel Processing)



Multiple Consumer Groups



Consumer Group Rules

Partitions vs Consumers in a Group:

Case 1: Consumers < Partitions

3 Partitions, 2 Consumers

P0, P1 → Consumer 1

P2 → Consumer 2

✓ Works, but uneven load

Case 2: Consumers = Partitions (IDEAL)

3 Partitions, 3 Consumers

P0 → Consumer 1

P1 → Consumer 2

P2 → Consumer 3

✓ Perfect distribution

Case 3: Consumers > Partitions

3 Partitions, 5 Consumers

P0 → Consumer 1

P1 → Consumer 2

P2 → Consumer 3

Consumer 4: IDLE !

Consumer 5: IDLE !

! Extra consumers are wasted!

7. Message Delivery Guarantees

Three Delivery Semantics

DELIVERY GUARANTEES

1. AT MOST ONCE (may lose messages)

- Fire and forget
- No retries
- Fastest, but unreliable
- Use for: Metrics, logs (loss acceptable)

2. AT LEAST ONCE (may duplicate messages) ← MOST COMMON

- Retries on failure
- Consumer processes, then commits
- Same message might be processed twice
- Use for: Most applications (with idempotent handling)

3. EXACTLY ONCE (no loss, no duplicates)

- Transactional producer + consumer
- Most complex, some overhead
- Use for: Financial transactions, critical data

Idempotent Producer

Problem: Network issues cause duplicate sends

Producer —msg—> Broker (success, but ack lost)

Producer —msg—> Broker (retry = DUPLICATE!)

Solution: Idempotent Producer (enable.idempotence=true)

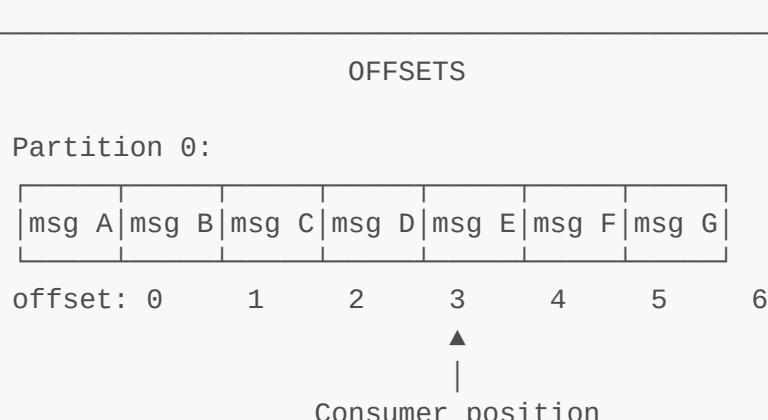
Producer —msg (seq=1)—> Broker ✓

Producer —msg (seq=1)—> Broker (deduplicated) ✓

Kafka tracks sequence numbers per producer to detect duplicates.

8. Offsets and Commits

What is an Offset?



Commit Strategies

COMMIT STRATEGIES

AUTO COMMIT (enable.auto.commit=true)

- Kafka commits automatically every 5 seconds
 - Simple but may lose messages on crash

Timeline:

On restart: Continues from offset 3 ✓

But what if:

[read msg1] [read msg2] [crash] (before auto commit)

On restart: Re-reads msg1, msg2 (duplicates!) !

MANUAL COMMIT (`enable.auto.commit=false`) ← RECOMMENDED

- You control when to commit
 - Commit AFTER successful processing

If crash before commit → message redelivered

If crash after commit → no duplicate

Part 2: Integration with Spring Boot

9. Project Setup

Dependencies (pom.xml)

```
<!-- Kafka -->
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>

<!-- JSON serialization -->
<dependency>
```

```
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
</dependency>
```

Docker Compose (Infrastructure)

```
# Zookeeper (Kafka needs this for coordination)
zookeeper:
  image: confluentinc/cp-zookeeper:7.6.1
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
  ports:
    - "2181:2181"

# Kafka Broker
kafka:
  image: confluentinc/cp-kafka:7.6.1
  depends_on:
    - zookeeper
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1 # 1 for dev, 3 for prod
  ports:
    - "9092:9092"
```

10. Configuration

application-dev.yml

```
spring:
  kafka:
    # Where to find Kafka
    bootstrap-servers: localhost:9092

    # Producer settings
    producer:
      key-serializer:
        org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
        org.springframework.kafka.support.serializer.JsonSerializer
      acks: all          # Wait for all replicas
      retries: 3         # Retry 3 times on failure
      properties:
        enable.idempotence: true # Prevent duplicates
        compression.type: snappy # Compress messages
```

```

# Consumer settings
consumer:
  group-id: purchasement-group
  key-deserializer:
    org.apache.kafka.common.serialization.StringDeserializer
  value-deserializer:
    org.springframework.kafka.support.serializer.JsonDeserializer
  auto-offset-reset: earliest # Start from beginning if no offset
  enable-auto-commit: false # Manual commit for reliability
  properties:
    spring.json.trusted.packages: "*" # Allow deserialization

# Listener settings
listener:
  ack-mode: manual-immediate # Manual acknowledgment
  concurrency: 3 # 3 consumer threads

```

KafkaConfig.java - What Each Setting Does

```

@Configuration
public class KafkaConfig {

    // =====
    // PRODUCER CONFIGURATION
    // =====

    @Bean
    public ProducerFactory<String, Object> producerFactory() {
        Map<String, Object> config = new HashMap<>();

        // Where is Kafka?
        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");

        // How to convert data to bytes?
        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        JsonSerializer.class);

        // Reliability: Wait for ALL replicas to acknowledge
        config.put(ProducerConfig.ACKS_CONFIG, "all");

        // Retry: Try 3 times if send fails
        config.put(ProducerConfig.RETRIES_CONFIG, 3);

        // Idempotence: Prevent duplicate messages
        config.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Performance: Batch messages together
        config.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384); // 16KB
    }
}

```

```
batch
    config.put(ProducerConfig.LINGER_MS_CONFIG, 5); // Wait
5ms for more msgs
    config.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy"); // Compress

    return new DefaultKafkaProducerFactory<>(config);
}

// =====
// CONSUMER CONFIGURATION
// =====

@Bean
public ConsumerFactory<String, Object> consumerFactory() {
    Map<String, Object> config = new HashMap<>();

    config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

    // Deserializers: Convert bytes back to objects
    config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

    // Consumer group name
    config.put(ConsumerConfig.GROUP_ID_CONFIG, "purchasement-group");

    // Where to start if no previous offset?
    config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

    // Manual commit (we control when to acknowledge)
    config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

    // Performance: Max messages per poll
    config.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);

    return new DefaultKafkaConsumerFactory<>(config,
        new StringDeserializer(),
        new JsonDeserializer<>());
}

// =====
// LISTENER CONTAINER (connects consumer to your methods)
// =====

@Bean
public ConcurrentKafkaListenerContainerFactory<String, Object>
kafkaListenerContainerFactory() {

    var factory = new ConcurrentKafkaListenerContainerFactory<String,
Object>();
    factory.setConsumerFactory(consumerFactory());
```

```
// Manual acknowledgment mode
factory.getContainerProperties()
    .setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

// Run 3 consumer threads
factory.setConcurrency(3);

// Error handler: Retry 3 times, then send to DLQ
factory.setCommonErrorHandler(errorHandler()));

    return factory;
}
}
```

11. Creating Topics

KafkaTopicConfig.java

```
@Configuration
public class KafkaTopicConfig {

    @Bean
    public KafkaAdmin kafkaAdmin() {
        Map<String, Object> config = new HashMap<>();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        return new KafkaAdmin(config);
    }

    // =====
    // Topics are auto-created when application starts
    // =====

    @Bean
    public NewTopic userCreatedTopic() {
        return TopicBuilder.name("user-created")
            .partitions(3)          // 3 partitions for parallelism
            .replicas(1)           // 1 replica (use 3 in production)
            .config("retention.ms", "604800000") // Keep messages for
7 days
            .build();
    }

    @Bean
    public NewTopic orderCreatedTopic() {
        return TopicBuilder.name("order-created")
            .partitions(6)          // More partitions = higher
throughput
            .replicas(1)
    }
}
```

```
        .config("retention.ms", "2592000000") // 30 days for orders
        .build();
    }

    // Dead Letter Queue - for failed messages
    @Bean
    public NewTopic dlqUserTopic() {
        return TopicBuilder.name("dlq-user-created")
            .partitions(1)           // Single partition for ordering
            .replicas(1)
            .config("retention.ms", "2592000000") // Keep DLQ longer
            .build();
    }
}
```

12. Building the Producer

KafkaProducerService.java

```
@Service
@RequiredArgsConstructor
public class KafkaProducerService {

    private final KafkaTemplate<String, Object> kafkaTemplate;

    // =====
    // ASYNC SEND - Fire and forget (with logging callback)
    // =====

    public void sendAsync(String topic, Object message) {
        // Send returns immediately, callback handles result later
        CompletableFuture<SendResult<String, Object>> future =
            kafkaTemplate.send(topic, message);

        future.whenComplete((result, exception) -> {
            if (exception != null) {
                log.error("Failed to send to {}: {}", topic,
                    exception.getMessage());
            } else {
                log.info("Sent to {} partition {} offset {}",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
            }
        });
    }

    // =====
    // ASYNC WITH KEY - Same key = same partition = ordered
    // =====
```

```
public void sendAsync(String topic, String key, Object message) {
    // Key ensures all messages with same key go to same partition
    // Example: All events for user-123 are ordered
    kafkaTemplate.send(topic, key, message);
}

// =====
// SYNC SEND - Wait for confirmation
// =====

public boolean sendSync(String topic, Object message) {
    try {
        // .get() blocks until Kafka confirms receipt
        SendResult<String, Object> result = kafkaTemplate
            .send(topic, message)
            .get(10, TimeUnit.SECONDS); // Wait max 10 seconds

        log.info("Sync send successful: offset {}", result.getRecordMetadata().offset());
        return true;
    } catch (Exception e) {
        log.error("Sync send failed: {}", e.getMessage());
        return false;
    }
}

// =====
// SEND TO SPECIFIC PARTITION
// =====

public void sendToPartition(String topic, int partition, String key,
Object message) {
    ProducerRecord<String, Object> record =
        new ProducerRecord<>(topic, partition, key, message);
    kafkaTemplate.send(record);
}

// =====
// BATCH SEND
// =====

public void sendBatch(String topic, List<?> messages) {
    messages.forEach(msg -> sendAsync(topic, msg));
    log.info("Batch of {} messages sent to {}", messages.size(),
topic);
}
```

13. Building Consumers

UserEventConsumer.java

```
@Service
@Slf4j
public class UserEventConsumer {

    // =====
    // BASIC LISTENER - With manual acknowledgment
    // =====

    @KafkaListener(
        topics = "user-created",           // Which topic to listen
        groupId = "user-service-group",    // Consumer group name
        containerFactory = "kafkaListenerContainerFactory"
    )
    public void handleUserCreated(
        @Payload UserEvent event,         // The message content
        @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
        @Header(KafkaHeaders.RECEIVED_PARTITION) int partition,
        @Header(KafkaHeaders.OFFSET) long offset,
        Acknowledgment acknowledgment) { // For manual commit

        log.info("Received: {} from {}-{} at offset {}", event, topic, partition, offset);

        try {
            // =====
            // YOUR BUSINESS LOGIC HERE
            // =====

            processUserCreated(event);

            // SUCCESS: Acknowledge the message
            // This commits the offset - message won't be redelivered
            acknowledgment.acknowledge();

        } catch (Exception e) {
            // FAILURE: Don't acknowledge
            // Error handler will retry, then send to DLQ
            log.error("Processing failed: {}", e.getMessage());
            throw e; // Let error handler manage it
        }
    }

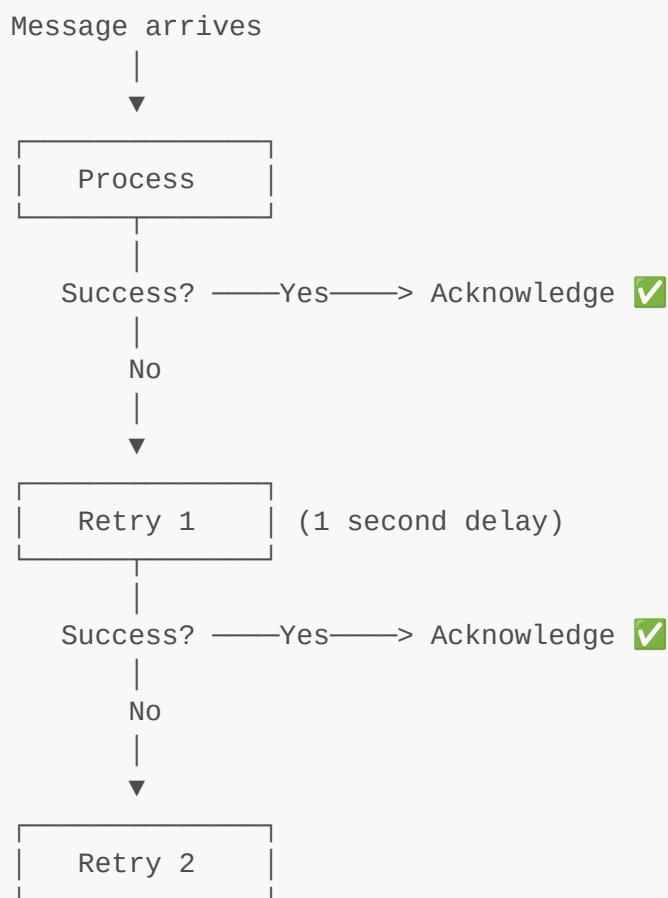
    private void processUserCreated(UserEvent event) {
        // Example: Save to database, send welcome email, etc.
        log.debug("Processing user: {}", event.getUsername());
    }
}
```

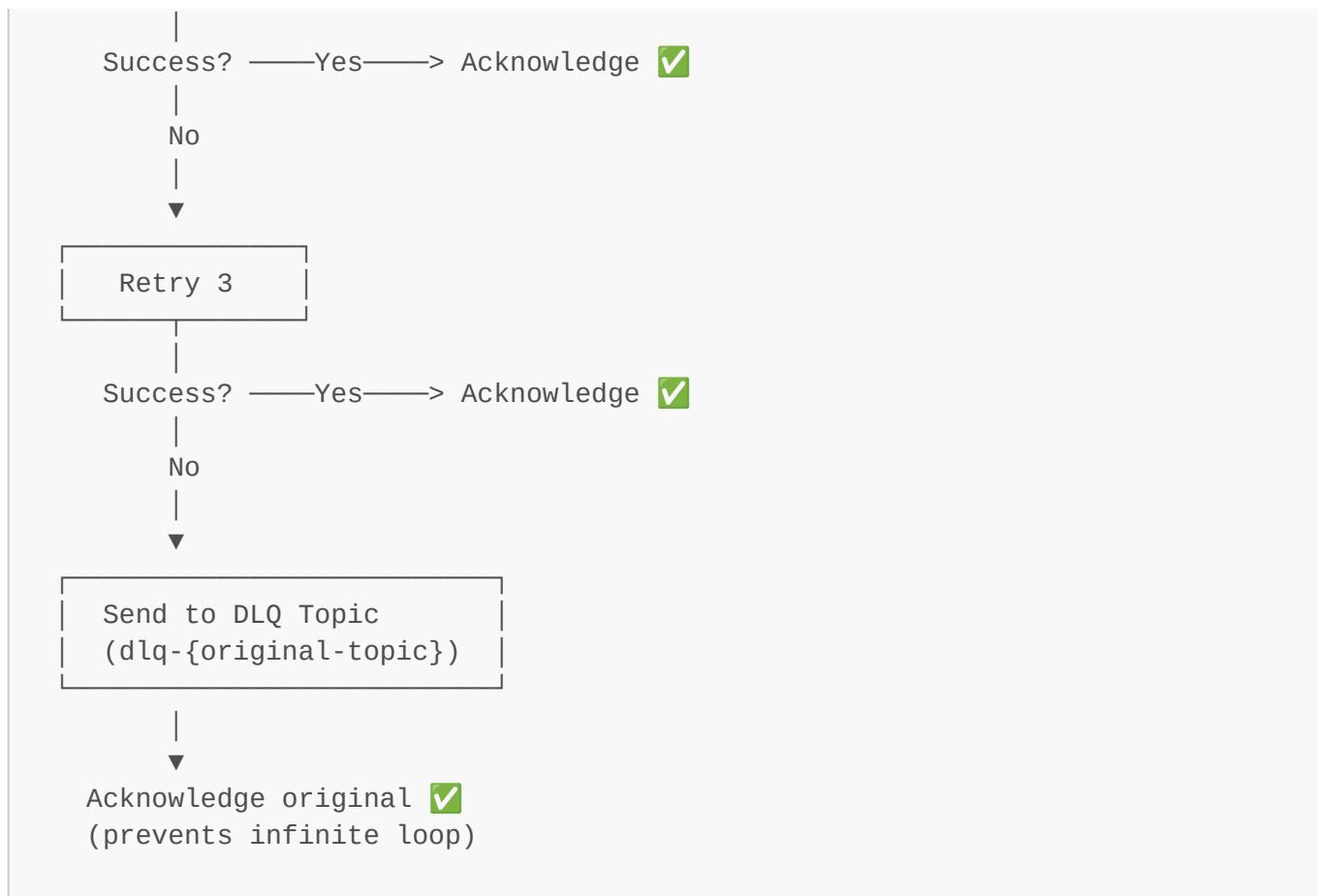
Batch Consumer (High Throughput)

```
@KafkaListener(  
    topics = "order-updated",  
    groupId = "order-service-group",  
    containerFactory = "batchKafkaListenerContainerFactory" // Different  
factory!  
)  
public void handleOrdersBatch(  
    @Payload List<OrderEvent> events, // List of messages  
    Acknowledgment acknowledgment) {  
  
    log.info("Received batch of {} orders", events.size());  
  
    // Process all messages  
    for (OrderEvent event : events) {  
        processOrder(event);  
    }  
  
    // Acknowledge entire batch at once  
    acknowledgment.acknowledge();  
}
```

14. Error Handling and DLQ

How Error Handling Works





Error Handler Configuration

```

@Bean
public CommonErrorHandler errorHandler(KafkaTemplate<String, Object>
kafkaTemplate) {

    // Dead Letter Queue publisher
    // Routes failed messages to: dlq-{original-topic}
    DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(
        kafkaTemplate,
        (record, exception) -> {
            String dlqTopic = "dlq-" + record.topic();
            log.error("Sending to DLQ: {} due to: {}", dlqTopic,
exception.getMessage());
            return new TopicPartition(dlqTopic, record.partition());
        }
);

    // Retry configuration: 3 attempts, 1 second apart
    FixedBackOff backOff = new FixedBackOff(1000L, 3L);

    DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

    // Log each retry attempt
    errorHandler.setRetryListeners((record, ex, deliveryAttempt) -> {

```

```
        log.warn("Retry {} for topic {} offset {}",  
            deliveryAttempt, record.topic(), record.offset());  
    });  
  
    return errorHandler;  
}
```

DLQ Consumer

```
@Service  
public class DeadLetterQueueConsumer {  
  
    @KafkaListener(  
        topics = "dlq-user-created",  
        groupId = "dlq-handler-group",  
        containerFactory = "dlqKafkaListenerContainerFactory" // No DLQ  
    for DLQ!  
    )  
    public void handleDLQ(ConsumerRecord<String, Object> record,  
                          Acknowledgment acknowledgment) {  
  
        log.error("== DEAD LETTER RECEIVED ==");  
        log.error("Topic: {}", record.topic());  
        log.error("Partition: {}", record.partition());  
        log.error("Offset: {}", record.offset());  
        log.error("Value: {}", record.value());  
  
        // Get error details from headers  
        record.headers().forEach(header -> {  
            log.error("Header {}: {}", header.key(), new  
String(header.value()));  
        });  
  
        // Options:  
        // 1. Log for manual investigation  
        // 2. Save to database for later retry  
        // 3. Send alert (Slack, email)  
        // 4. Attempt automatic fix and republish  
  
        acknowledgment.acknowledge();  
    }  
}
```

Part 3: Code Walkthrough

15. Project Structure

```

src/main/java/com/distributed_system/purchasement/
|
|   common/
|   |   config/
|   |   |   KafkaConfig.java          # Producer, Consumer, Error Handler
|   |   config
|   |   |   KafkaTopicConfig.java    # Auto topic creation
|   |
|   |   constant/
|   |   |   KafkaTopics.java        # All topic names as constants
|   |
|   |   controller/
|   |   |   KafkaTestController.java # REST endpoints for testing
|   |
|   |   event/                  # Event classes (DTOS)
|   |   |   BaseEvent.java          # Common fields: eventId, timestamp,
etc.
|   |   |   UserEvent.java         # User-related events
|   |   |   OrderEvent.java        # Order-related events
|   |   |   PaymentEvent.java     # Payment-related events
|   |
|   |   service/
|   |   |   kafka/
|   |   |   |   KafkaProducerService.java      # Sending messages
|   |   |   |   UserEventConsumer.java       # Handles user events
|   |   |   |   OrderEventConsumer.java     # Handles order events
|   |   |   |   PaymentEventConsumer.java   # Handles payment events
|   |   |   |   DeadLetterQueueConsumer.java # Handles failed messages
|
|   entity/
|   |   User.java
|

```

16. Event Design

BaseEvent - Common Fields

```

public abstract class BaseEvent {

    private String eventId;           // Unique ID (UUID)
    private String eventType;         // "USER_CREATED", "ORDER_COMPLETED"
    private LocalDateTime timestamp; // When event was created
    private String source;           // Service that created it
    private String correlationId;    // Links related events together
    private int version;             // For schema evolution

    // Initialize common fields
    public void initializeMetadata(String eventType, String source) {
        this.eventId = UUID.randomUUID().toString();
        this.eventType = eventType;
    }
}

```

```
    this.timestamp = LocalDateTime.now();
    this.source = source;
    this.version = 1;
    if (this.correlationId == null) {
        this.correlationId = UUID.randomUUID().toString();
    }
}
}
```

UserEvent - With Factory Methods

```
public class UserEvent extends BaseEvent {

    private Long userId;
    private String username;
    private String email;
    private int age;
    private String action; // CREATED, UPDATED, DELETED

    // Factory methods - easy to create correct events

    public static UserEvent created(Long userId, String username,
                                    String email, int age) {
        UserEvent event = UserEvent.builder()
            .userId(userId)
            .username(username)
            .email(email)
            .age(age)
            .action("CREATED")
            .build();
        event.initializeMetadata("USER_CREATED", "purchasement-service");
        return event;
    }

    public static UserEvent deleted(Long userId) {
        UserEvent event = UserEvent.builder()
            .userId(userId)
            .action("DELETED")
            .build();
        event.initializeMetadata("USER_DELETED", "purchasement-service");
        return event;
    }
}
```

Using Events

```
// In your controller or service:

// Create a user event
```

```
UserEvent event = UserEvent.created(123L, "john", "john@email.com", 25);

// Send it
kafkaProducerService.sendEventAsync(KafkaTopics.USER_CREATED, event);

// The event will have:
// - eventId: "abc-123-def-456"
// - eventType: "USER_CREATED"
// - timestamp: "2024-01-15T10:30:00"
// - source: "purchasement-service"
// - correlationId: "xyz-789"
// - userId: 123
// - username: "john"
// - email: "john@email.com"
// - age: 25
// - action: "CREATED"
```

17. Producer Service Explained

Different Ways to Send Messages

```
@Service
public class KafkaProducerService {

    // =====
    // 1. SIMPLE ASYNC - For most cases
    // =====

    public void sendAsync(String topic, Object message) {
        kafkaTemplate.send(topic, message)
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    log.error("Send failed: {}", ex.getMessage());
                } else {
                    log.info("Sent to partition {} offset {}",
                            result.getRecordMetadata().partition(),
                            result.getRecordMetadata().offset());
                }
            });
    }

    // Usage:
    // kafkaProducerService.sendAsync("user-created", userEvent);

    // =====
    // 2. WITH KEY - When ordering matters
    // =====

    public void sendAsync(String topic, String key, Object message) {
```

```
// Same key = same partition = messages processed in order
kafkaTemplate.send(topic, key, message);
}

// Usage (all user-123 events go to same partition, processed in
order):
// kafkaProducerService.sendAsync("user-events", "user-123", event1);
// kafkaProducerService.sendAsync("user-events", "user-123", event2);

// =====
// 3. SYNC - When you need confirmation
// =====

public boolean sendSync(String topic, Object message) {
    try {
        // Blocks until Kafka confirms
        kafkaTemplate.send(topic, message).get(10, TimeUnit.SECONDS);
        return true;
    } catch (Exception e) {
        return false;
    }
}

// Usage:
// boolean success = kafkaProducerService.sendSync("payment-completed",
event);
// if (!success) { /* handle failure */ }

// =====
// 4. WITH CALLBACK - Custom success/error handling
// =====

public void sendWithCallback(String topic, Object message,
                             Runnable onSuccess,
                             Consumer<Throwable> onError) {
    kafkaTemplate.send(topic, message)
        .whenComplete((result, ex) -> {
            if (ex != null) {
                onError.accept(ex);
            } else {
                onSuccess.run();
            }
        });
}

// Usage:
// kafkaProducerService.sendWithCallback(
//     "orders",
//     orderEvent,
//     () -> emailService.sendConfirmation(), // On success
//     error -> alertService.notify(error) // On error
```

```
// );  
}
```

18. Consumer Service Explained

Anatomy of a Consumer

```
@Service  
@Slf4j  
public class UserEventConsumer {  
  
    @KafkaListener(  
        topics = "user-created",                      // Topic to listen to  
        groupId = "user-service-group",                // Consumer group  
        containerFactory = "kafkaListenerContainerFactory"  
    )  
    public void handleUserCreated(  
        //  
        // Parameters you can inject (all optional except @Payload)  
        //  
  
        @Payload UserEvent event,                      // The message content  
  
        @Header(KafkaHeaders.RECEIVED_TOPIC)           String topic,                      // Topic name  
  
        @Header(KafkaHeaders.RECEIVED_PARTITION)         int partition,                   // Partition number  
  
        @Header(KafkaHeaders.OFFSET)                    long offset,                     // Message offset  
  
        @Header(KafkaHeaders.RECEIVED_TIMESTAMP)         long timestamp,                 // When message was produced  
  
        @Header(value = "correlationId", required = false) String correlationId,          // Custom header  
  
        Acknowledgment acknowledgment,                  // For manual commit  
  
        Consumer<String, Object> consumer            // Raw consumer (rarely  
needed)  
    ) {  
        log.info("Received event from topic={} partition={} offset={}",  
            topic, partition, offset);  
  
        try {  
            //  
            // Your business logic  
            //
```

```
        processUser(event);

        // =====
        // SUCCESS: Acknowledge (commit offset)
        // This tells Kafka: "I'm done with this message"
        // =====

        acknowledgment.acknowledge();

    } catch (Exception e) {
        // =====
        // FAILURE: Don't acknowledge, throw exception
        // Error handler will retry, then send to DLQ
        // =====

        log.error("Processing failed: {}", e.getMessage());
        throw e;
    }
}

}
```

Event Chaining - One Event Triggers Another

```
@Service
public class PaymentEventConsumer {

    @Autowired
    private KafkaProducerService producerService;

    @KafkaListener(topics = "payment-completed", groupId = "payment-service")
    public void handlePaymentCompleted(PaymentEvent event,
                                        Acknowledgment ack) {
        log.info("Payment completed for order {}", event.getOrderId());

        // Payment succeeded → Trigger order completion
        OrderEvent orderComplete = OrderEvent.completed(
            event.getOrderId(),
            event.getUserId(),
            event.getAmount()
        );

        // Maintain correlation ID for tracing
        orderComplete.setCorrelationId(event.getCorrelationId());

        // Send order completion event
        producerService.sendEventAsync("order-completed", orderComplete);

        ack.acknowledge();
    }
}
```

```
@KafkaListener(topics = "payment-failed", groupId = "payment-service")
public void handlePaymentFailed(PaymentEvent event,
                                Acknowledgment ack) {
    log.warn("Payment failed for order {}: {}", event.getOrderId(), event.getFailureReason());

    // Payment failed → Trigger order cancellation
    OrderEvent orderCancel = OrderEvent.cancelled(
        event.getOrderId(),
        event.getUserId()
    );
    orderCancel.setCorrelationId(event.getCorrelationId());

    producerService.sendEventAsync("order-cancelled", orderCancel);

    ack.acknowledge();
}
}
```

19. Testing the System

Start Infrastructure

```
# Start Kafka and Zookeeper
docker-compose -f docker-compose-infrastructure.yml up -d kafka zookeeper

# Verify they're running
docker ps | grep -E "kafka|zookeeper"
```

Start Your Application

```
# Run Spring Boot app
./mvnw spring-boot:run

# Or if using IDE, just run PurchasementApplication.java
```

Test Commands

```
# =====
# USER EVENTS
# =====

# Create user
curl -X POST "http://localhost:7777/kafka/test/user/create?"
```

```
username=john&email=john@test.com&age=25"

# Expected console output:
# INFO - Message sent to topic user-created partition 0 offset 0
# INFO - Received USER_CREATED event: eventId=abc-123...
# INFO - Successfully processed USER_CREATED event: john

# Update user
curl -X POST "http://localhost:7777/kafka/test/user/update?
userId=123&username=john_updated&email=john@test.com&age=26"

# Delete user
curl -X POST "http://localhost:7777/kafka/test/user/delete?userId=123"

# =====
# ORDER EVENTS
# =====

# Create order
curl -X POST "http://localhost:7777/kafka/test/order/create?
userId=123&totalAmount=99.99"

# Cancel order
curl -X POST "http://localhost:7777/kafka/test/order/cancel?
orderId=123&userId=456"

# Complete order
curl -X POST "http://localhost:7777/kafka/test/order/complete?
orderId=123&userId=456&totalAmount=99.99"

# =====
# PAYMENT EVENTS (with event chaining)
# =====

# Initiate payment
curl -X POST "http://localhost:7777/kafka/test/payment/initiate?
orderId=123&userId=456&amount=99.99&method=CREDIT_CARD"

# Complete payment (triggers ORDER_COMPLETED automatically)
curl -X POST "http://localhost:7777/kafka/test/payment/complete?
paymentId=123&orderId=456&transactionId=TXN123"

# Expected output:
# INFO - Received PAYMENT_COMPLETED event: paymentId=123, orderId=456
# INFO - Triggered ORDER_COMPLETED event for orderId=456
# INFO - Received ORDER_COMPLETED event: orderId=456

# Fail payment (triggers ORDER_CANCELLED automatically)
curl -X POST "http://localhost:7777/kafka/test/payment/fail?
paymentId=123&orderId=456&reason=Insufficient%20funds"
```

```
# =====
# ADVANCED TESTS
# =====

# Synchronous send (waits for Kafka confirmation)
curl -X POST "http://localhost:7777/kafka/test-sync?topic=user-created&message=Hello"

# Batch send (10 messages at once)
curl -X POST "http://localhost:7777/kafka/test/batch?count=10"

# Full e-commerce flow (User → Order → Payment → Completion)
curl -X POST "http://localhost:7777/kafka/test/full-flow?username=john&email=john@test.com"

# Expected output for full-flow:
# INFO - Message sent to user-created
# INFO - Message sent to order-created
# INFO - Message sent to payment-initiated
# INFO - Message sent to payment-completed
# INFO - Received PAYMENT_COMPLETED → Triggered ORDER_COMPLETED
# INFO - Received ORDER_COMPLETED

# =====
# TEST DEAD LETTER QUEUE
# =====

# Trigger DLQ (sends invalid message)
curl -X POST "http://localhost:7777/kafka/test/trigger-dlq"

# Expected output:
# WARN - Retry attempt 1 for topic user-created
# WARN - Retry attempt 2 for topic user-created
# WARN - Retry attempt 3 for topic user-created
# ERROR - Sending to DLQ topic: dlq-user-created
# ERROR - == DEAD LETTER RECEIVED ==
# ERROR - Topic: dlq-user-created
# ERROR - Value: {invalid=true, test=This message should fail}
```

Verify with Kafka CLI (Optional)

```
# List all topics
docker exec kafka kafka-topics --list --bootstrap-server localhost:9092

# Check messages in a topic
docker exec kafka kafka-console-consumer \
--bootstrap-server localhost:9092 \
--topic user-created \
--from-beginning
```

```
# Check consumer groups
docker exec kafka kafka-consumer-groups \
  --bootstrap-server localhost:9092 \
  --list

# Check consumer group lag (how far behind)
docker exec kafka kafka-consumer-groups \
  --bootstrap-server localhost:9092 \
  --describe --group user-service-group
```

Summary

What You Learned

1. Kafka Fundamentals

- Brokers, Clusters, Topics, Partitions
- Producers and Consumers
- Consumer Groups and Offsets
- Delivery Guarantees

2. Spring Boot Integration

- Configuration for Producer and Consumer
- Auto topic creation
- Error handling with retries and DLQ

3. Code Implementation

- Event design with BaseEvent
- Producer service with multiple send methods
- Consumers with manual acknowledgment
- Event chaining between services

Key Takeaways

Concept	Remember
Partitions	More partitions = more parallelism
Keys	Same key = same partition = ordered
Consumer Groups	Each group gets ALL messages
Manual Commit	Acknowledge AFTER successful processing
DLQ	Failed messages go here for investigation
Correlation ID	Links related events for tracing

Next Steps

- Add monitoring (Prometheus + Grafana)
- Implement schema registry
- Add integration tests
- Set up multi-broker cluster for production
- Add transaction support for exactly-once semantics