

# 16. 메시징과 통합

## Kafka 연동

### • Producer, Consumer

#### ✖ 1. Producer란?

메시지를 생성해서 브로커(중간자)에게 전달하는 역할

예: 주문 완료 시 메시지 발행 → "OrderCreated"

✓ 비동기 처리 시작점

✓ Kafka, RabbitMQ, Redis 등 메시지 브로커에 `send/publish`

#### 💡 2. Consumer란?

브로커로부터 메시지를 수신하여 처리하는 역할

예: 결제 시스템이 주문 메시지를 수신하고 처리 시작

✓ 비동기 처리 종착점

✓ 메시지를 받아서 DB 저장, API 호출 등 다양한 작업 수행

#### 🔄 3. 전체 메시지 흐름

```
1 [Producer]
2   ↓ publish
3 [Broker] (Kafka, RabbitMQ 등)
4   ↓ subscribe
5 [Consumer]
```

예시:

- 주문 서비스 → "OrderCreated" 이벤트 발행 (Producer)
- 결제 서비스가 해당 이벤트 수신 후 결제 처리 (Consumer)

#### ⚙ 4. Spring Boot 기반 구성 예시

여기선 **Kafka** 기준으로 예시 설명할게

RabbitMQ도 거의 동일한 구조

## ✓ 의존성 추가 (Gradle)

```
1 implementation("org.springframework.kafka:spring-kafka")
```

## ✓ application.yml 설정

```
1 spring:
2   kafka:
3     bootstrap-servers: localhost:9092
4     consumer:
5       group-id: my-service
6       auto-offset-reset: earliest
7       key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
8       value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
9     producer:
10      key-serializer: org.apache.kafka.common.serialization.StringSerializer
11      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

## 5. Producer 구현 예시

```
1 @Service
2 @RequiredArgsConstructor
3 public class OrderProducer {
4
5     private final KafkaTemplate<String, String> kafkaTemplate;
6
7     public void publishOrderCreated(Long orderId) {
8         String message = String.format("{\"orderId\": %d}", orderId);
9         kafkaTemplate.send("order-created-topic", message);
10    }
11 }
```

## 6. Consumer 구현 예시

```
1 @Component
2 public class OrderConsumer {
3
4     @KafkaListener(topics = "order-created-topic", groupId = "payment-service")
5     public void consume(String message) {
6         System.out.println("✓ 주문 생성 이벤트 수신: " + message);
7         // 파싱 → 처리 로직 수행
8     }
9 }
```

## 7. 실무 적용 시 고려사항

항목	Producer 측	Consumer 측
메시지 중복	idempotent하게 처리	✅ 필수
메시지 포맷	JSON, Avro, Protobuf	스키마 통일
실패 처리	Retry, Dead Letter Topic	예외/로그 처리
순서 보장	Partition 관리	Kafka: 같은 key → 같은 partition
비동기 트랜잭션	DB + 메시지 전송 순서 보장?	Outbox 패턴 필요

## 8. Producer-Consumer 패턴의 장점

장점	설명
비동기 처리	응답 시간 단축, 병렬 처리 가능
서비스 분리	강한 의존 제거 (느슨한 연결)
확장성	Consumer를 여러 개 확장 가능
장애 격리	한쪽 다운돼도 메시지는 보존됨

## 9. 실무 설계 팁

설계 항목	권장 방식
메시지 스키마	DTO → JSON 변환 + 명확한 필드 정의
로그	발행 성공/실패 로깅 필수
재처리	실패 시 재시도 또는 DLQ 구성
트랜잭션	Kafka → DB 동시에 다룰 땐 Outbox 패턴 고려
테스트	Embedded Kafka 또는 WireMock 조합 추천

## ✅ 마무리 요약표

항목	설명
Producer	메시지 발행자 ( <code>kafkaTemplate.send()</code> )
Consumer	메시지 수신자 ( <code>@kafkaListener</code> )
브로커	Kafka, RabbitMQ 등 메시지 중계

항목	설명
포맷	JSON/AVRO 권장 (명확한 스키마)
실행 흐름	발행 → 큐 저장 → 수신/처리
트랜잭션 처리	메시지/DB 분리 시 Outbox 패턴 검토

## • @KafkaListener

### ✖ 1. 개념

Spring Kafka에서 Kafka 메시지를 수신하고 처리하기 위해 사용하는 어노테이션.

- 내부적으로 KafkaConsumer를 등록하여 메시지를 **자동 수신**
- @KafkaListener가 선언된 메서드는 **브로커로부터 메시지를 비동기로 처리함**

### ■ 2. 기본 사용법

```

1 @KafkaListener(topics = "order-created", groupId = "order-consumer")
2 public void handleOrder(String message) {
3     System.out.println("✅ 주문 메시지 수신: " + message);
4 }

```

파라미터	의미
topics	수신할 Kafka 토픽 이름
groupId	Consumer Group 이름 (같은 그룹은 파티션을 나눠 가짐)

### 🔄 3. Kafka 메시지 구조 바인딩

#### ✅ 문자열 메시지

```

1 @KafkaListener(topics = "simple-topic")
2 public void handle(String message) { ... }

```

#### ✅ JSON 메시지를 객체로 매핑

```

1 @KafkaListener(topics = "user-topic", containerFactory =
2     "userKafkaListenerContainerFactory")
3 public void handleUser(UserDto dto) {
4     System.out.println("👤 사용자 이벤트 수신: " + dto.getName());
5 }

```

```

1 @Bean
2 public ConcurrentKafkaListenerContainerFactory<String, UserDto>
   userKafkaListenerContainerFactory() {
3     var factory = new ConcurrentKafkaListenerContainerFactory<String, UserDto>();
4     factory.setConsumerFactory(new DefaultKafkaConsumerFactory<>(consumerProps(),
       new StringDeserializer(), new JsonSerializer<>(UserDto.class)));
5     return factory;
6 }

```

## 4. Consumer Group 동작 구조

그룹	파티션 수	인스턴스 수	처리 방식
A	3	3	각 인스턴스가 하나의 파티션을 처리
A	3	1	하나의 인스턴스가 모든 파티션 처리
A	3	4	하나는 대기 상태 (파티션 수보다 인스턴스가 많으면 일부는 idle)

✔ 동일 Group ID를 가진 인스턴스는 **파티션 단위로 분산** 처리됨

## 5. 병렬 처리 (Concurrency)

```

1 @Bean
2 public ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory()
   {
3     var factory = new ConcurrentKafkaListenerContainerFactory<>();
4     factory.setConcurrency(3); // 최대 3개 스레드로 병렬 처리
5     return factory;
6 }

```

병렬성은 **파티션 수 이하로만 유효**

Kafka는 각 파티션을 하나의 Consumer가 독점

## 6. 예외 처리 전략

✔ **try-catch 사용**

```

1 @KafkaListener(topics = "payment")
2 public void handle(String message) {
3     try {
4         // 처리 로직
5     } catch (Exception e) {
6         log.error(" ! 메시지 처리 실패: {}", message, e);
7         // 재처리 큐에 적재 or DLQ 전송
8     }
9 }

```

## ✅ 글로벌 에러 핸들러 설정

```
1 @Bean
2 public ConcurrentKafkaListenerContainerFactory<String, String>
   kafkaListenerContainerFactory() {
3     var factory = new ConcurrentKafkaListenerContainerFactory<String, String>();
4     factory.setErrorHandler((thrownException, data) -> {
5         log.error(" ! 예외 발생: {}", data, thrownException);
6     });
7     return factory;
8 }
```

## ✅ 재시도 (Retry + Backoff)

```
1 factory.setRetryTemplate(retryTemplate());
2
3 private RetryTemplate retryTemplate() {
4     var retry = new RetryTemplate();
5     retry.setRetryPolicy(new SimpleRetryPolicy(3)); // 최대 3번 재시도
6     retry.setBackOffPolicy(new FixedBackOffPolicy() {{
7         setBackOffPeriod(2000); // 2초 간격
8     }});
9     return retry;
10 }
```

## 🔒 7. 보안 / 역직렬화 실패 대비

- 메시지 형식이 예상과 다르면 `DeserializationException` 발생 → 반드시 예외 처리
- `JsonDeserializer` 사용 시: `trusted.packages=*` 설정 필요 (또는 명시 지정)

```
1 spring:
2   kafka:
3     consumer:
4       properties:
5         spring.json.trusted.packages: "*"

```

## ✅ 마무리 요약표

항목	설명
<code>@KafkaListener(topics = ...)</code>	메시지 수신 메서드 등록
<code>groupId</code>	컨슈머 그룹 설정 (파티션 분배 기준)
<code>containerFactory</code>	메시지 역직렬화 방식 지정

항목	설명
병렬 처리	<code>concurrency</code> 수로 컨슈머 동시성 조절
예외 처리	try-catch, 글로벌 에러핸들러, 재시도 설정 가능
DTO 수신	<code>JsonDeserializer</code> 로 객체 매핑 가능
트랜잭션 연계	<code>@KafkaListener</code> 는 DB 트랜잭션과 분리됨 → 수동 제어 가능

## RabbitMQ 연동

### • @RabbitListener

#### ✖ 1. @RabbitListener란?

Spring AMQP에서 **RabbitMQ** 큐에 바인딩된 메시지를 비동기로 수신하는 어노테이션

- ✓ `@KafkaListener` 와 유사하지만, AMQP 기반으로 동작
- ✓ Spring Boot + RabbitMQ에서 메시지 기반 소비 처리할 때 사용

#### ⚙ 2. 기본 사용 예시

```

1 @RabbitListener(queues = "order.queue")
2 public void handleOrder(String message) {
3     System.out.println("✓ 메시지 수신: " + message);
4 }

```

파라미터	설명
<code>queues</code>	수신할 큐 이름
<code>bindings</code>	(선택) Exchange/Queue/Binding 설정 동시 정의

#### ■ 3. 필수 의존성

##### ✓ Gradle

```

1 implementation("org.springframework.boot:spring-boot-starter-amqp")

```

## 📁 4. application.yml 설정

```
1 spring:
2   rabbitmq:
3     host: localhost
4     port: 5672
5     username: guest
6     password: guest
7     listener:
8       simple:
9         retry:
10          enabled: true
11          max-attempts: 3
12          initial-interval: 1000
```

## 📦 5. 메시지 객체 바인딩 (JSON → DTO)

### ✅ 메시지 DTO

```
1 public class OrderMessage {
2   private Long orderId;
3   private String userEmail;
4   // getters/setters
5 }
```

### ✅ 리스너

```
1 @RabbitListener(queues = "order.queue")
2 public void receiveOrder(OrderMessage message) {
3   System.out.println("📦 주문 수신: " + message.getOrderId());
4 }
```

➡ Jackson 메시지 변환 자동 적용됨 (`MappingJackson2MessageConverter`)

## 🔄 6. 큐/익스체인지/바인딩 동시 선언

```
1 @RabbitListener(
2   bindings = @QueueBinding(
3     value = @Queue(value = "order.queue", durable = "true"),
4     exchange = @Exchange(value = "order.exchange", type = "topic"),
5     key = "order.*"
6   )
7 )
8 public void consume(String message) {
9   System.out.println("📩 수신: " + message);
10 }
```

✅ 코드 내에서 RabbitMQ 구성 자동 등록 가능



## 7. 동시성 처리 (Concurrency)

```
1 spring:
2   rabbitmq:
3     listener:
4       simple:
5         concurrency: 3
6         max-concurrency: 10
```

컨슈머 동시 처리 스레드 수 설정 (파티션 개념 없음)

## 8. 예외 처리

### ✓ 기본: 예외 발생 시 재시도 or DLQ

- `listener.simple.retry.enabled=true` 설정 시 자동 재시도
- 재시도 초과 시 메시지는 기본적으로 버려짐 or DLQ 구성 필요

### ✓ try-catch로 수동 처리

```
1 @RabbitListener(queues = "alert.queue")
2 public void consumewithError(String msg) {
3     try {
4         // 비즈니스 처리
5     } catch (Exception e) {
6         log.error(" ! 처리 실패: {}", msg, e);
7         // 실패 시 재처리 로직 or 실패 로그 저장
8     }
9 }
```

### ✓ Dead Letter Queue 설정 예시

```
1 spring:
2   rabbitmq:
3     listener:
4       simple:
5         default-requeue-rejected: false
```

그리고 Queue 선언 시:

```

1 @Bean
2 public Queue orderQueue() {
3     return QueueBuilder.durable("order.queue")
4         .withArgument("x-dead-letter-exchange", "dlx.exchange")
5         .withArgument("x-dead-letter-routing-key", "dlx.routing")
6         .build();
7 }

```

## 🧠 9. 실무 설계 팁

항목	권장 전략
메시지 포맷	JSON + DTO 명확한 구조로 통일
큐 설계	도메인 기반 큐 이름 설계 ( <code>order.queue</code> , <code>payment.queue</code> )
바인딩 설정	코드로 함께 정의 or YAML로 선언
예외 처리	재시도 → DLQ or 보상 로직 구성
다중 큐 수신	<code>@RabbitListener(queues = {"a", "b"})</code> 가능
순서 보장	RabbitMQ는 큐 단위 순서를 보장하므로 파티션 필요 없음

## ✅ 마무리 요약

항목	설명
<code>@RabbitListener</code>	메시지를 큐에서 비동기 수신
큐 지정	<code>queues = "..."</code> 또는 <code>bindings</code> 로 Exchange 포함 선언
메시지 타입	DTO 자동 매핑 (Jackson)
동시성	<code>concurrency</code> , <code>max-concurrency</code> 설정
예외 처리	자동 재시도 + DLQ 구성 가능
실무 활용	주문, 알림, 이벤트 처리 등 메시지 기반 처리

## 메시지 컨버터

### ✂ 1. 메시지 컨버터란?

메시지 바이트/텍스트를 객체(자바 POJO)로 직렬화·역직렬화하는 구성 요소

방향	동작
Producer → Broker	객체 → 바이트/문자열 (직렬화)

방향	동작
Broker → Consumer	바이트/문자열 → 객체 (역직렬화) ✅ 핵심

Spring에서는 메시지 컨버터를 통해

- ✅ JSON → DTO 자동 매핑,
- ✅ DTO → JSON 자동 직렬화가 가능함

## 📦 2. 주요 메시지 컨버터 종류

컨버터 클래스	설명
<code>StringMessageConverter</code>	문자열 ↔ 문자열 (기본)
<code>MappingJackson2MessageConverter</code>	JSON ↔ DTO 자동 매핑 (AMQP, WebSocket 등)
<code>JsonMessageConverter</code> (spring-kafka)	Kafka 전용 Jackson 기반 JSON 컨버터
<code>ByteArrayMessageConverter</code>	바이트 배열 ↔ 객체
커스텀 컨버터	사용자 정의 직렬화/역직렬화 방식 구현

## 📘 3. Spring Kafka에서 메시지 컨버터 등록

### ✅ JSON → DTO 매핑 예시

```

1 @Bean
2 public ConcurrentKafkaListenerContainerFactory<String, MyEvent>
   kafkaListenerContainerFactory(
3     ConsumerFactory<String, MyEvent> consumerFactory) {
4
5     ConcurrentKafkaListenerContainerFactory<String, MyEvent> factory = new
       ConcurrentKafkaListenerContainerFactory<>();
6     factory.setConsumerFactory(consumerFactory);
7     factory.setMessageConverter(new StringJsonMessageConverter()); // 핵심!
8     return factory;
9 }

```

➡ `@KafkaListener` 는 JSON 문자열을 DTO로 자동 변환

```

1 @KafkaListener(topics = "my-topic", containerFactory = "kafkaListenerContainerFactory")
2 public void handle(MyEvent event) {
3     System.out.println("📧 수신: " + event.getName());
4 }

```



## 4. Spring AMQP(RabbitMQ)에서 메시지 컨버터 설정

```

1  @Bean
2  public MessageConverter jsonMessageConverter() {
3      return new Jackson2JsonMessageConverter();
4  }
5
6  @Bean
7  public RabbitTemplate rabbitTemplate(ConnectionFactory cf) {
8      RabbitTemplate template = new RabbitTemplate(cf);
9      template.setMessageConverter(jsonMessageConverter());
10     return template;
11 }
12
13 @Bean
14 public SimpleRabbitListenerContainerFactory
15     rabbitListenerContainerFactory(ConnectionFactory cf) {
16     SimpleRabbitListenerContainerFactory factory = new
17     SimpleRabbitListenerContainerFactory();
18     factory.setConnectionFactory(cf);
19     factory.setMessageConverter(jsonMessageConverter()); // 수신 시 컨버터
20     return factory;
21 }

```

➡ RabbitMQ에서도 @RabbitListener 에서 DTO 수신 가능

```

1  @RabbitListener(queues = "order.queue")
2  public void handle(OrderEvent event) {
3      System.out.println("📦 주문 수신: " + event.getOrderid());
4  }

```



## 5. 커스텀 메시지 컨버터 예시

```

1  public class CustomUpperCaseConverter implements MessageConverter {
2
3      @Override
4      public Message toMessage(Object object, MessageProperties props) throws
5      MessageConversionException {
6          String payload = ((String) object).toUpperCase();
7          return new Message(payload.getBytes(), props);
8      }
9
10     @Override
11     public Object fromMessage(Message message) throws MessageConversionException {
12         return new String(message.getBody()).toLowerCase();
13     }
14 }

```

➡ 문자 변환, 암호화, 압축 등도 가능

## 6. 예외 처리

- JSON 파싱 실패 → `DeserializationException`, `MessageConversionException` 발생
- Spring에서는 `ErrorHandler` 나 try-catch에서 처리

```
1 @KafkaListener(topics = "user")
2 public void handleSafe(String json) {
3     try {
4         UserDto dto = objectMapper.readValue(json, UserDto.class);
5     } catch (JsonProcessingException e) {
6         log.error(" ! 메시지 역직렬화 실패", e);
7     }
8 }
```

## 7. 실무 설계 팁

항목	권장 전략
포맷	JSON 통일 + Jackson 컨버터 사용
메시지 구조	DTO 설계 시 명확한 필드와 타입 사용
에러 처리	파싱 실패 시 로그 저장 + DLQ 전송 고려
통합	Kafka, RabbitMQ, WebSocket 등 동일 컨버터 방식 유지
테스트	Consumer 단위 테스트 시 컨버터까지 포함하여 검증

## 마무리 요약

항목	설명
메시지 컨버터	메시지를 DTO로 자동 변환해주는 컴포넌트
Kafka 전용	<code>StringJsonMessageConverter</code>
AMQP 전용	<code>Jackson2JsonMessageConverter</code>
등록 위치	<code>ListenerFactory</code> 또는 <code>Template</code> 에 명시
실무 활용	JSON → DTO, DTO → JSON 자동 처리
예외 처리	JSON 파싱 실패 시 <code>DeserializationException</code> 발생

# 메시지 재처리와 리트라이 전략

## ❧ 1. 왜 메시지 재처리가 중요한가?

메시지 소비 중 예외가 발생하면 그 메시지를 다시 시도하거나, 실패 메시지를 별도 큐로 분리(DLQ)해야 시스템이 안정적이고 견고해짐

✅ 예:

- DB 일시적 장애 → 메시지 재처리 필요
- 외부 API 타임아웃 → 재시도 후 성공 가능성 있음
- 비정상 메시지 → DLQ로 보내고 로그/모니터링

## ⚙️ 2. Kafka vs RabbitMQ 기본 동작 차이

항목	Kafka	RabbitMQ
메시지 저장	디스크에 영구 저장	메모리/디스크
메시지 삭제	Consumer가 commit 시	Consumer가 ack 시
실패 처리	기본 재시도 없음	기본 자동 재전송
DLQ	별도 구성 필요	공식 DLX 설정 존재
재시도 제어	수동 구현 ( <code>SeekToCurrent</code> , retry topic)	자동 or 수동 가능

## ■ 3. Spring Kafka: 재시도 + 에러 핸들링

✅ 방식 1: 수동 `try-catch`

```
1 @KafkaListener(topics = "order")
2 public void handle(String msg) {
3     try {
4         // 처리
5     } catch (Exception e) {
6         log.error(" ! 처리 실패: {}", msg);
7         // 재시도 큐로 이동 or 무시
8     }
9 }
```

## ✅ 방식 2: 컨테이너 레벨 리트라이 설정

```
1 @Bean
2 public ConcurrentKafkaListenerContainerFactory<String, String>
   kafkaListenerContainerFactory() {
3     var factory = new ConcurrentKafkaListenerContainerFactory<>();
4     factory.setRetryTemplate(retryTemplate());
5     factory.setRecoveryCallback(context -> {
6         String msg = (String) context.getAttribute("record");
7         log.error("❌ 최종 실패 메시지 처리: {}", msg);
8         return null;
9     });
10    return factory;
11 }
```

```
1 private RetryTemplate retryTemplate() {
2     RetryTemplate retry = new RetryTemplate();
3
4     retry.setRetryPolicy(new SimpleRetryPolicy(3)); // 최대 3회
5     retry.setBackOffPolicy(new FixedBackOffPolicy() {{
6         setBackOffPeriod(2000); // 2초 간격
7     }});
8
9     return retry;
10 }
```

## ✅ Kafka 고급 전략: Retry Topic + DLQ

- Spring Kafka v2.7+ 부터 `@RetryableTopic`, `@DltHandler` 지원

```
1 @RetryableTopic(
2     attempts = "3",
3     backoff = @Backoff(delay = 2000),
4     dltTopicsSuffix = "-dlt"
5 )
6 @KafkaListener(topics = "payment")
7 public void handle(String msg) {
8     ...
9 }
10
11 @DltHandler
12 public void handleDlt(String dltMessage) {
13     log.error("💀 DLQ 도달 메시지: {}", dltMessage);
14 }
```

➡ 자동으로 `payment-retry`, `payment-dlt` 토픽이 생성되고 재시도/실패 메시지 분리됨

## 4. Spring RabbitMQ 재시도 전략

### ✓ 기본 자동 재전송 (ack 안 하면 재전달)

```
1 @RabbitListener(queues = "order.queue")
2 public void handle(String msg) {
3     throw new RuntimeException("실패!"); // ack 안됨 → 자동 재시도
4 }
```

### ✓ 리트라이 구성 (application.yml)

```
1 spring:
2   rabbitmq:
3     listener:
4       simple:
5         retry:
6           enabled: true
7           max-attempts: 3
8           initial-interval: 1000
9           multiplier: 2.0
```

➡ 기본적으로 재시도 후 실패 시 message discard

### ✓ DLQ 설정

```
1 @Bean
2 public Queue orderQueue() {
3     return QueueBuilder.durable("order.queue")
4       .withArgument("x-dead-letter-exchange", "dlx.exchange")
5       .withArgument("x-dead-letter-routing-key", "dlx.routing")
6       .build();
7 }
```

## 5. 실무 재시도 설계 전략

항목	설계 기준
단순 예외	재시도 (ex: DB connection timeout)
비정상 메시지	DLQ로 분리 후 수동 확인
리트라이 간격	Fixed or Exponential Backoff
최대 횟수	3~5회, 이후 DLQ로 이동
DLQ 모니터링	Slack, email, dashboard 연동
처리 보장	Idempotent 처리 필수 (재전송 시 중복 방지)



## ✅ 마무리 요약표

항목	Kafka	RabbitMQ
기본 재시도	❌ (수동)	✅
리스너 재시도 설정	RetryTemplate or @RetryableTopic	spring.rabbitmq.listener.simple.retry.*
DLQ 구성	수동 토픽 구성 or @DltHandler	x-dead-letter-exchange, x-dead-letter-routing-key
Backoff	Fixed/Exponential	YAML에서 설정
메시지 보존	디스크 기반 (재시도 주체는 consumer)	큐 내 보존 후 재전송

## Dead Letter Queue (DLQ)

### ❗ 1. DLQ란?

메시지가 소비 중 지속적으로 실패하거나 거부되었을 때,  
해당 메시지를 별도의 큐(토픽)로 이동시켜 격리/보존하는 구조

- ✅ 재처리 실패 메시지를 분리하고,
- ✅ 정상 흐름을 유지하며
- ✅ 장애 분석, 보상 처리, 추적이 가능하게 함

### 🔄 2. 동작 흐름 요약

```
1 Producer → Main Queue/Topic
2     ↓ (실패 N회, 예외, nack 등)
3     Dead Letter Queue (DLQ)
```

예:

- 3번 재시도 후 실패 → DLQ로 전송
- JSON 파싱 실패 → DLQ 전송 후 알림

### ⚙️ 3. Kafka 기반 DLQ 구성

#### ✅ 방법 1: @RetryableTopic + @DltHandler (Spring Kafka 2.7+)

```
1 @RetryableTopic(
2     attempts = "3",
3     backoff = @Backoff(delay = 2000),
4     dltTopicSuffix = "-dlt"
5 )
6 @KafkaListener(topics = "order")
```

```

7 public void consume(OrderMessage msg) {
8     throw new RuntimeException("처리 실패");
9 }

```

➡ 자동으로 다음 토픽 생성:

- `order-retry-0`, `order-retry-1` (재시도용)
- `order-dlt` (DLQ)

## ✅ DLQ 처리

```

1 @DltHandler
2 public void handleDlt(OrderMessage msg) {
3     log.error("💀 DLQ 메시지 도착: {}", msg);
4     // 알림 전송, DB 저장, 수동 보상
5 }

```

## ✅ 방법 2: 수동 DLQ Producer 전송

```

1 @KafkaListener(topics = "order")
2 public void handle(String msg) {
3     try {
4         ...
5     } catch (Exception ex) {
6         kafkaTemplate.send("order-dlt", msg); // DLQ 수동 전송
7     }
8 }

```

## 🐇 4. RabbitMQ 기반 DLQ 구성

### ✅ Queue 설정: `x-dead-letter-exchange`

```

1 @Bean
2 public Queue mainQueue() {
3     return QueueBuilder.durable("order.queue")
4         .withArgument("x-dead-letter-exchange", "dlt.exchange")
5         .withArgument("x-dead-letter-routing-key", "order.dlt")
6         .build();
7 }
8
9 @Bean
10 public Queue deadLetterQueue() {
11     return QueueBuilder.durable("order.dlt").build();
12 }
13
14 @Bean
15 public DirectExchange dltExchange() {
16     return new DirectExchange("dlt.exchange");
17 }

```

```

18
19 @Bean
20 public Binding dlqBinding() {
21     return BindingBuilder.bind(deadLetterQueue())
22         .to(dlExchange()).with("order.dlq");
23 }

```

➡ 메시지 소비 중 예외 발생 시 DLQ로 자동 전송됨

## 5. Spring Rabbit에서 DLQ 자동 전송 조건

조건	설명
예외 발생	@RabbitListener 에서 Exception throw
Ack 실패	수동 ack 실패 시
TTL 만료	메시지 지연 후 만료되면 DLQ로 이동
Queue full	큐 초과 시 정책 설정에 따라 DLQ 전송

## 6. DLQ 재처리 전략

전략	설명
수동 재처리	DLQ 메시지를 읽고, 원래 큐로 재전송
대시보드 + 승인	관리자가 승인 시 재전송 or Drop
자동 재처리	일정 시간 후 polling하여 재시도

```

1 @Scheduled(fixedDelay = 60000)
2 public void reconsumeDlq() {
3     List<Message> dlqMsgs = dlqRepo.fetch();
4     for (Message msg : dlqMsgs) {
5         mainProducer.send(msg.getPayload()); // 원래 큐로 재전송
6     }
7 }

```

## 7. DLQ 모니터링 구성

도구	설명
Spring Boot Actuator	Kafka, RabbitMQ 상태 확인 가능
Prometheus + Grafana	DLQ 메시지 수, 에러 수 대시보드
Log Alert	DLQ 수신 로그 Slack 연동

도구	설명
Dead Letter Audit	DB 테이블에 DLQ 메시지 저장 + 트래킹

## 8. 실무 설계 팁

항목	권장 전략
DLQ 이름 규칙	<code>order.dlq</code> , <code>user.dlt</code> , <code>topic-name-dlq</code>
메시지 포맷	DTO + 실패 이유 포함 ( <code>reason</code> , <code>timestamp</code> )
DLQ 보존 기간	TTL 설정 + 수동 or 자동 정리
알림 시스템	DLQ 도착 시 Slack, Email, JIRA 연동
보상 처리	관리자 승인 + 재처리 API or Batch 스케줄러




## 마무리 요약

항목	설명
DLQ 목적	실패 메시지 격리, 정상 흐름 보호, 사후 분석
Kafka	<code>@RetryableTopic</code> + <code>@DltHandler</code> , or 수동 DLQ
RabbitMQ	<code>x-dead-letter-exchange</code> , <code>x-dead-letter-routing-key</code>
재처리	수동 or 자동 재전송 구조 구성
모니터링	대시보드, 로그, 알림 연계 필수
실무 기준	DLQ는 “실패를 기록하고 다시 성공할 기회를 주는 구조”

# Spring Integration 기본 구조

## 1. Spring Integration이란?

다양한 시스템, 프로토콜, 메시지를 Spring 기반으로 **모듈화된 방식으로 통합**하기 위한 프레임워크.

-  EIP(Enterprise Integration Patterns) 구현
-  MQ, HTTP, JMS, TCP/UDP, FTP, Mail, File 등 다양한 **입출력 연동** 가능
-  메시지를 중심으로 **비동기/동기 이벤트 흐름**을 설계

## 2. 왜 사용하는가?

- 시스템 간 **비동기 데이터 흐름**을 구성
- Kafka, RabbitMQ, File, REST API, DB 등 다양한 채널을 통합
- **오케스트레이션 / 파이프라인 처리 / 메시지 라우팅 / 변환** 등 구현
- 이벤트 드리븐 아키텍처를 **코드가 아닌 DSL 구성으로 설계**

## 3. Spring Integration 핵심 개념 구조

1 | [Inbound Adapter] → [Message Channel] → [Message Handler or Router] → [Outbound Adapter]

구성 요소	설명
<b>Message</b>	실제 데이터 (payload) + 메타 정보 (headers)
<b>Channel</b>	메시지를 전달하는 통로 (DirectChannel, QueueChannel, PublishSubscribeChannel)
<b>Gateway</b>	외부 요청을 시스템 메시지로 변환
<b>Adapter</b>	외부 시스템과 연동 (e.g. Kafka, Mail, FTP)
<b>Transformer</b>	메시지를 다른 형식으로 변환
<b>Router</b>	조건에 따라 메시지 흐름을 분기
<b>Filter</b>	메시지를 필터링 (drop or pass)
<b>Service Activator</b>	실제 비즈니스 로직 처리기
<b>Bridge</b>	채널 간 메시지를 연결

## 4. 전체 흐름 예시 (예: REST → Kafka)

1 | HTTP → Inbound Gateway → Channel → Transformer → Kafka Outbound Adapter

1. 클라이언트 요청 (/send)
2. 메시지 생성 → Channel로 전달
3. Transformer가 메시지 가공
4. Kafka로 전송됨

## 🔧 5. 기본 설정 예시 (Java DSL)

```
1  @Configuration
2  @EnableIntegration
3  public class IntegrationFlowConfig {
4
5      @Bean
6      public IntegrationFlow sampleFlow() {
7          return IntegrationFlows.from(Http.inboundGateway("/send"))
8              .transform(String.class, String::toUpperCase) // 대문자 변환
9              .handle(m -> System.out.println("🔔 수신 메시지: " + m.getPayload()))
10             .get();
11     }
12 }
```

➡ `/send` 로 POST하면 메시지가 대문자로 변환되어 출력됨

## 📦 6. 주요 컴포넌트 정리

### ✅ Channel (메시지 통로)

```
1  @Bean
2  public MessageChannel inputChannel() {
3      return new DirectChannel();
4  }
```

- `DirectChannel`: 싱글 소비자, 동기 처리
- `QueueChannel`: 비동기 대기 가능
- `PublishSubscribeChannel`: 브로드캐스트

### ✅ Transformer (메시지 변환)

```
1  .transform(String.class, msg -> "Hello, " + msg)
```

### ✅ Router (조건 분기)

```
1  .route(String.class, msg -> msg.contains("VIP") ? "vipChannel" : "normalChannel")
```

### ✅ Service Activator (핸들러)

```
1  .handle(MyMessageHandler.class, "process")
```

또는 람다로 처리

```
1 | .handle(msg -> processMessage(msg.getPayload()))
```

## 💡 7. 실전 사용 예시

시나리오	구성
주문 수신 후 Kafka 전송	<code>Http</code> → <code>Transformer</code> → <code>Kafka.outboundChannelAdapter()</code>
DB → Polling → MQ	<code>JdbcInboundAdapter</code> → <code>Channel</code> → <code>RabbitAdapter</code>
파일 업로드 처리	<code>FileInboundAdapter</code> → <code>Filter</code> → <code>ServiceActivator</code>
이메일 수신 자동 처리	<code>MailInboundAdapter</code> → <code>Transformer</code> → <code>SlackNotifier</code>

## ✅ 마무리 요약표

구성 요소	역할
Message	payload + headers
Channel	메시지 경로 ( <code>Direct</code> , <code>Queue</code> , <code>PublishSubscribe</code> )
Gateway	외부 호출 ↔ 메시지 변환
Adapter	외부 시스템 연동 (Kafka, FTP, Mail 등)
Transformer	메시지 형식 변환
Filter	메시지 통과 여부 결정
Router	조건에 따라 분기 처리
Service Activator	실제 로직 수행 위치