12. 메시지 기반 비동기 처리

Spring Events

Spring Events는 **ApplicationEventPublisher 기반의 비동기 이벤트 처리 시스템**으로,
애플리케이션 내부 컴포넌트 간 **결합도를 낮추고 유연한 확장성**을 제공하기 위한 **Observer 패턴** 기반의 기능입니다.

Spring에서는 **이벤트 발행 → 리스너 수신 → 작업 처리**의 구조로
비즈니스 로직의 흐름을 느슨하게 연결할 수 있으며, **동기/비동기** 방식 모두 지원됩니다.

✅ 1. Spring Events 개요

구성 요소	설명
Event	전달하고자 하는 정보 객체 (일반 POJO 또는 `ApplicationEvent`)
Publisher	이벤트를 발행하는 쪽 (`ApplicationEventPublisher`)
Listener	이벤트를 수신하여 처리하는 쪽 (`@EventListener`)

✅ 2. 기본 사용 구조

📌 1. 이벤트 클래스 정의

```
public class UserRegisteredEvent {
    private final Long userId;

    public UserRegisteredEvent(Long userId) {
        this.userId = userId;
    }

    public Long getUserId() {
        return userId;
    }
}
```

> POJO 형태로도 충분하며, 더 이상 `extends ApplicationEvent`는 필요 없음

📌 2. 이벤트 발행

```
@Service
public class UserService {

    private final ApplicationEventPublisher eventPublisher;

    public UserService(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
```

```
    }

    public void registerUser(String name) {
        // 사용자 등록 로직...
        Long newUserId = 123L;

        // 이벤트 발행
        eventPublisher.publishEvent(new UserRegisteredEvent(newUserId));
    }
}
```

📌 3. 이벤트 수신 및 처리

```
@Component
public class WelcomeEmailListener {

    @EventListener
    public void handleUserRegistered(UserRegisteredEvent event) {
        System.out.println("회원가입 환영 메일 발송: " + event.getUserId());
        // 이메일 발송 로직
    }
}
```

✅ 3. 동기 vs 비동기 처리

📌 동기 처리 (기본값)

- `publishEvent()` 호출 → 리스너 메서드가 **즉시 실행됨**
- 예외 발생 시 발행자에게 전파됨

📌 비동기 처리

1. `@EnableAsync` 활성화

```
@Configuration
@EnableAsync
public class AsyncConfig { }
```

1. 리스너에 `@Async` 추가

```
@Async
@EventListener
public void handleAsync(UserRegisteredEvent event) {
    // 별도 스레드에서 실행
}
```

> 실행은 `TaskExecutor` 스레드풀에서 이루어짐

✅ 4. 리스너 조건 필터링

```
@EventListener(condition = "#event.userId > 100")
public void handleFiltered(UserRegisteredEvent event) {
    // userId가 100 초과일 때만 실행
}
```

✅ 5. 트랜잭션 이벤트 리스너

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void handleAfterCommit(UserRegisteredEvent event) {
    // 트랜잭션 커밋 후 실행 (DB 반영 완료 시점)
}
```

Phase 옵션	설명
`BEFORE_COMMIT`	커밋 직전에 실행
`AFTER_COMMIT`	커밋 완료 후 실행 (가장 일반적)
`AFTER_ROLLBACK`	롤백 후 실행
`AFTER_COMPLETION`	커밋/롤백 관계없이 항상 실행

✅ 6. 커스텀 ApplicationEvent 사용 (구버전 스타일)

```
public class LegacyEvent extends ApplicationEvent {
    public LegacyEvent(Object source) {
        super(source);
    }
}
```

> Spring 4.2 이후는 **POJO 이벤트 클래스**로 충분

✅ 7. 실무 활용 예시

상황	설명
회원가입 후 후처리	이메일 발송, 포인트 지급 등
주문 완료 이벤트	메시지 큐 전송, 관리자 알림 등
로그 수집	이벤트로 비동기 로깅 처리
도메인 이벤트	DDD 패턴 적용 시 Entity → 이벤트 발행

✅ 8. 테스트에서 이벤트 확인

```
@Autowired
private ApplicationEventPublisher publisher;

@Test
void testEvent() {
    publisher.publishEvent(new UserRegisteredEvent(123L));
}
```

- 이벤트 리스너가 잘 작동하는지 `@MockBean` 또는 로그로 검증 가능

✅ 9. 장점 요약

항목	설명
낮은 결합도	서비스끼리 직접 의존하지 않아도 됨
확장 용이	이벤트 리스너만 추가하면 후처리 기능 확장
비동기 처리	사용량 많은 작업을 병렬로 실행 가능
트랜잭션 연동	`@TransactionalEventListener` 로 DB 처리 완료 후 작업 가능

✅ 결론 요약

항목	설명
발행자	`ApplicationEventPublisher.publishEvent()` 사용
수신자	`@EventListener`, `@Async`, `@TransactionalEventListener`
이벤트 클래스	POJO 또는 `ApplicationEvent` 상속
기본 실행	동기 방식 (예외 전파 O)
비동기 실행	`@EnableAsync` + `@Async` 사용
실무 용도	후처리 분리, 트랜잭션 후 처리, 비동기 알림 등

비동기 처리: `@Async`, `@EnableAsync`

Spring에서의 비동기 처리 기능은 `@Async` 애노테이션과 `@EnableAsync` 설정을 통해
간단하게 멀티스레드 기반의 비동기 로직을 구현할 수 있도록 지원됩니다.
이는 특히 **시간이 오래 걸리는 작업(예: 이메일 발송, 파일 처리, 외부 API 호출)** 등을
별도 스레드로 실행하여 응답 성능을 향상시킬 수 있습니다.

✅ 1. 기본 개념

항목	설명
`@Async`	메서드를 **비동기 스레드에서 실행**하도록 지정
`@EnableAsync`	Spring 비동기 기능을 **전역 활성화**하는 설정
스레드 실행 방식	기본적으로 `SimpleAsyncTaskExecutor`, 실무에선 ThreadPool 설정 권장
반환 타입	`void`, `Future<T>`, `CompletableFuture<T>`, `ListenableFuture<T>` 등 가능

✅ 2. 기본 사용 방법

📌 1. 설정 클래스에 `@EnableAsync` 추가

```
@Configuration
@EnableAsync
public class AsyncConfig { }
```

📌 2. 비동기로 실행할 메서드에 `@Async` 추가

```
@Service
public class MailService {

    @Async
    public void sendWelcomeEmail(String email) {
        System.out.println("[메일 발송 시작] Thread: " +
Thread.currentThread().getName());

        try {
            Thread.sleep(3000); // 모의 지연
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("[메일 발송 완료] 수신자: " + email);
    }
}
```

📌 3. 호출부 예시

```
mailService.sendWelcomeEmail("kim@example.com");
```

> `@Async` 메서드는 **프록시 기반으로 동작**하기 때문에
> 반드시 **다른 클래스에서 호출해야** 정상 작동함

✅ 3. 반환값이 있는 비동기 메서드

```
@Async
public CompletableFuture<String> process() {
    return CompletableFuture.completedFuture("처리 완료");
}
```

📌 호출부에서 `get()`으로 결과 확인

```
CompletableFuture<String> result = service.process();
String value = result.get(); // 블로킹됨
```

✅ 4. 커스텀 Executor 설정 (ThreadPoolTaskExecutor)

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "asyncExecutor")
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4); // 최소 스레드 수
        executor.setMaxPoolSize(10); // 최대 스레드 수
        executor.setQueueCapacity(100); // 대기 큐 용량
        executor.setThreadNamePrefix("Async-Thread-");
        executor.initialize();
        return executor;
    }
}
```

📌 특정 Executor를 명시적으로 사용하고 싶을 때

```
@Async("asyncExecutor")
public void taskA() {
    ...
}
```

✅ 5. 예외 처리

- `void` 반환형인 경우 예외는 호출부에서 `catch` 불가
- `CompletableFuture`를 사용하면 예외 전달 가능

```java
@Async
public CompletableFuture<String> process() {
    try {
        // 작업 수행
    } catch (Exception e) {
        return CompletableFuture.failedFuture(e);
    }
}
```

✅ 6. 실무 예시

용도	설명
이메일 발송	`sendEmail()`을 비동기 실행하여 API 응답 속도 개선
영상/이미지 처리	업로드 후 별도 스레드에서 처리
로그 기록	사용자 요청과 별개로 비동기 처리
외부 API 호출	네트워크 지연 발생 작업 분리

✅ 7. 주의사항

항목	설명
내부 호출 ❌	같은 클래스 안에서 `this.asyncMethod()` → **비동기 작동 안함**
트랜잭션과 혼용	`@Transactional` 내부에서 `@Async` 호출 시 주의 (프록시 분리 필요)
Bean 등록 필수	`@EnableAsync`, `ThreadPoolTaskExecutor` 설정 권장
테스트 시	`@SpringBootTest`에서는 `@Async` 정상 작동하지만, 단순 단위 테스트는 동기 실행됨

✅ 8. 로그 기반 확인 팁

```java
System.out.println(Thread.currentThread().getName());
```

→ 비동기 스레드가 **Main Thread와 다르면 정상 작동 중**

✅ 결론 요약

항목	설명
비동기 메서드	`@Async`
비동기 활성화	`@EnableAsync`

항목	설명
커스텀 스레드풀	`ThreadPoolTaskExecutor`
반환값 처리	`CompletableFuture`, `Future`
실무 적용	알림, 비동기 처리, 작업 분산
주의사항	**프록시 기반 동작**, 내부 메서드 호출 X

RabbitMQ / Kafka 연동

Spring Boot에서 **RabbitMQ**와 **Kafka**는 메시지 기반의 비동기 통신을 위해 자주 사용되는 **메시지 브로커**입니다.
이 둘은 목적과 특성이 다르며, Spring에서는 각각을 위한 **전용 스타터와 설정 방식**을 제공합니다.

✅ RabbitMQ vs Kafka 간단 비교

항목	RabbitMQ	Kafka
메시지 처리 모델	**큐 기반 (Message Queue)**	**로그 기반 (Message Log)**
주요 목적	**작업 분산, 트랜잭션 처리**	**대용량 이벤트 스트리밍, 로그 수집**
메시지 보관	소비되면 삭제 (ack)	기본적으로 로그 유지 (n일)
소비 방식	1:1 또는 1:N (Fanout 등)	1:N (Consumer Group 단위)
신뢰성	고도화된 메시지 보장 (ack/nack, 재전송)	빠른 처리, 높은 처리량 우선
대표 사용	**마이크로서비스 간 RPC, 이벤트 처리**	**데이터 파이프라인, 실시간 분석, 로그 처리**

✅ 1. RabbitMQ 연동 (Spring AMQP)

📌 의존성 (Gradle)

```
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

📌 application.yml 설정 예시

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    virtual-host: /
```

📌 메시지 전송 (Producer)

```
@Service
public class RabbitProducer {

    private final RabbitTemplate rabbitTemplate;

    public RabbitProducer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    public void sendMessage(String message) {
        rabbitTemplate.convertAndSend("my.exchange", "my.routing.key", message);
    }
}
```

📌 메시지 수신 (Consumer)

```
@Component
public class RabbitConsumer {

    @RabbitListener(queues = "my.queue")
    public void receive(String message) {
        System.out.println("RabbitMQ 수신: " + message);
    }
}
```

📌 Queue/Exchange/Binding 설정

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue queue() {
        return new Queue("my.queue", true);
    }

    @Bean
    public DirectExchange exchange() {
        return new DirectExchange("my.exchange");
    }

    @Bean
    public Binding binding() {
        return BindingBuilder.bind(queue()).to(exchange()).with("my.routing.key");
    }
}
```

✅ 2. Kafka 연동 (Spring for Apache Kafka)

📌 의존성 (Gradle)

```
implementation 'org.springframework.boot:spring-boot-starter-kafka'
```

📌 application.yml 설정 예시

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: my-group
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

📌 메시지 전송 (Producer)

```
@Service
public class KafkaProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public KafkaProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void send(String topic, String message) {
        kafkaTemplate.send(topic, message);
    }
}
```

📌 메시지 수신 (Consumer)

```
@Component
public class KafkaConsumer {

    @KafkaListener(topics = "my-topic", groupId = "my-group")
    public void listen(String message) {
        System.out.println("Kafka 수신: " + message);
    }
}
```

✅ 3. JSON 메시지 처리 예시 (객체 직렬화)

Kafka 설정

```
spring.kafka:
  producer.value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
  consumer.value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
  consumer.properties.spring.json.trusted.packages: "*"
```

Kafka DTO 전송

```
public class UserEvent {
    private Long id;
    private String name;
}
```

✅ 4. 메시지 보장 전략 (예시)

기능	RabbitMQ	Kafka
메시지 손실 방지	`ack`, `durable queue`, `manualAck`	`acks=all`, `replication`
재시도	`retry`, `dead-letter exchange`	`retry topic`, `DLT`, `seek`
순서 보장	큐 단위	파티션 단위

✅ 5. 실무 활용 예

용도	RabbitMQ	Kafka
사용자 알림	✅	가능
이메일 큐	✅	가능
실시간 로그 분석	❌	✅
대용량 데이터 파이프라인	❌	✅
RPC (Request/Response)	✅	❌
보안 메시지 처리	✅	일부 구현 필요

✅ 6. 테스트 도구

도구	설명
RabbitMQ UI	`http://localhost:15672` (기본 guest/guest)
Kafka CLI	`kafka-console-consumer`, `kafka-console-producer`
Embedded Kafka (테스트용)	`@EmbeddedKafka`
TestContainers	`RabbitMQContainer`, `KafkaContainer`

✅ 결론 요약

항목	RabbitMQ	Kafka
주 사용 목적	큐, 메시지 분배	스트리밍, 로그 처리
Spring 연동	`spring-boot-starter-amqp`	`spring-boot-starter-kafka`
메시지 모델	큐 기반 (Ack 필수)	로그 기반 (Offset)
장점	신뢰성, 즉시 처리	고속, 고가용성, 저장
실무 적용	이벤트, 비동기 알림	로그 파이프라인, 실시간 분석

메시지 리스너 및 큐 구성

Spring Boot에서 메시지 기반 아키텍처를 사용할 때,
메시지 리스너(Message Listener)와 **큐(Queue) 구성**은 핵심 구성 요소입니다.
아래는 **RabbitMQ**와 **Kafka**를 기준으로 메시지 리스닝 및 큐/토픽 구성을 **실무 수준으로 정리한 가이드**입니다.

✅ 1. RabbitMQ 메시지 리스너 및 큐 구성

📌 1-1. 의존성 (Spring AMQP)

```
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

📌 1-2. 메시지 수신 리스너 (Consumer)

```
@Component
public class RabbitConsumer {

    @RabbitListener(queues = "my.queue")
    public void receive(String message) {
        System.out.println("수신 메시지: " + message);
    }
}
```

키워드	설명
`@RabbitListener`	해당 큐로 전달되는 메시지를 수신
`queues`	연결할 Queue 이름 (미리 선언 필요)

📌 1-3. 큐/익스체인지/바인딩 설정

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue queue() {
        return new Queue("my.queue", true); // durable
    }

    @Bean
    public DirectExchange exchange() {
        return new DirectExchange("my.exchange");
    }

    @Bean
    public Binding binding() {
        return BindingBuilder.bind(queue())
            .to(exchange())
            .with("my.routing.key");
    }
}
```

구성요소	설명
`Queue`	메시지가 저장되는 저장소
`Exchange`	메시지를 라우팅하는 허브 (`direct`, `topic`, `fanout`)
`Binding`	큐와 익스체인지를 연결 (라우팅키 필요)

📌 1-4. 메시지 자동 변환 (JSON → DTO)

```
@RabbitListener(queues = "my.queue")
public void receive(UserEvent event) {
    System.out.println("User ID: " + event.getUserId());
}
```

> Spring은 Jackson을 통해 JSON을 DTO로 자동 역직렬화함
> 단, `messageConverter` 설정 필요 없이 동작하는 경우도 있음

📌 1-5. Ack/Nack 수동 처리 (고급)

```
@RabbitListener(queues = "my.queue", ackMode = "MANUAL")
public void listen(Message message, Channel channel) throws IOException {
    try {
        String body = new String(message.getBody());
        // 처리 성공
        channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);
    } catch (Exception e) {
        // 실패 시 재처리 or 폐기
        channel.basicNack(message.getMessageProperties().getDeliveryTag(), false,
true);
    }
}
```

✅ 2. Kafka 메시지 리스너 및 토픽 구성

📌 2-1. 의존성

```
implementation 'org.springframework.boot:spring-boot-starter-kafka'
```

📌 2-2. 메시지 수신 리스너

```
@Component
public class KafkaConsumer {

    @KafkaListener(topics = "user-topic", groupId = "user-group")
    public void consume(String message) {
        System.out.println("Kafka 수신: " + message);
    }
}
```

속성	설명
`topics`	구독할 토픽 이름
`groupId`	동일 그룹 내에선 메시지를 1개만 처리 (로드밸런싱 효과)

📌 2-3. DTO 수신 (JSON → 객체)

```
@KafkaListener(topics = "user-topic", groupId = "user-group", containerFactory =
"userKafkaListenerContainerFactory")
public void consume(UserEvent user) {
    System.out.println("User ID: " + user.getUserId());
}
```

```java
@Bean
public ConcurrentKafkaListenerContainerFactory<String, UserEvent>
userKafkaListenerContainerFactory(
    ConsumerFactory<String, UserEvent> consumerFactory) {
    ConcurrentKafkaListenerContainerFactory<String, UserEvent> factory = new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    return factory;
}
```

→ Jackson 기반의 역직렬화를 위해 설정 필요

📌 2-4. Kafka 토픽 생성 (관리자 API 사용 시)

```java
@Bean
public NewTopic topic() {
    return TopicBuilder.name("user-topic")
        .partitions(3)
        .replicas(1)
        .build();
}
```

- Kafka는 기본적으로 동적 토픽 생성을 지원하지만, 명시적으로 생성하는 것도 좋음
- 파티션 수 = 병렬 소비 가능 수

📌 2-5. Offset 수동 커밋 처리 (고급)

```
spring.kafka.consumer.enable-auto-commit: false
```

```java
@KafkaListener(topics = "topic", containerFactory = "manualAckFactory")
public void listen(String message, Acknowledgment ack) {
    try {
        // 처리
        ack.acknowledge(); // 수동 커밋
    } catch (Exception e) {
        // 예외 처리
    }
}
```

✅ 실무 구성 전략 요약

항목	RabbitMQ	Kafka
메시지 리스너	`@RabbitListener`	`@KafkaListener`
큐 구성	`@Bean Queue`, `Exchange`, `Binding`	`@Bean NewTopic`

항목	RabbitMQ	Kafka
DTO 수신	JSON 자동 역직렬화	`containerFactory` 필요
병렬성	리스너당 다수 소비자 구성 가능	파티션 기반 자동 병렬 처리
수동 Ack	`Channel.basicAck()`	`Acknowledgment.acknowledge()`
트랜잭션 처리	`RabbitTransactionManager`	`@KafkaListener` + Tx 설정

✅ 결론 요약

항목	설명
메시지 수신	`@RabbitListener`, `@KafkaListener`로 선언
큐/토픽 구성	`@Configuration` 내에서 `@Bean`으로 등록
수신 메시지 처리	기본은 String, DTO도 가능 (역직렬화 설정 필요)
고급 처리	수동 Ack, 병렬 설정, DLT 구성 등 확장 가능
공통 이점	서비스 간 비동기 연동, 확장성, 모듈 간 결합도 제거

메시지 직렬화 전략 (JSON, AVRO)

Spring Boot 기반의 메시지 기반 시스템에서 **메시지 직렬화 전략**은
성능, 확장성, 상호운용성, 유지보수성에 큰 영향을 미칩니다.
가장 대표적인 직렬화 방식으로는 **JSON**과 **Apache Avro**가 있으며,
이들은 주로 Kafka, RabbitMQ 등의 브로커와 함께 사용됩니다.

✅ 1. JSON 직렬화

📌 개요

항목	설명
형식	사람이 읽을 수 있는 텍스트
장점	간단함, 디버깅 쉬움, 거의 모든 언어에서 지원
단점	**바이트 크기 큼**, **스키마 없음**, 타입 불명확성

📌 Spring Kafka에서 JSON 직렬화 설정

의존성

```
implementation 'org.springframework.kafka:spring-kafka'
```

---

application.yml 설정

```
spring.kafka:
  producer:
    value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
  consumer:
    value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
    properties:
      spring.json.trusted.packages: "*"
```

DTO 클래스 예시

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserEvent {
    private Long id;
    private String name;
}
```

KafkaTemplate 사용

```
kafkaTemplate.send("user-topic", new UserEvent(1L, "kim"));
```

📌 수신 측 리스너

```
@KafkaListener(topics = "user-topic", groupId = "user-group")
public void consume(UserEvent event) {
    System.out.println("수신 이벤트: " + event.getName());
}
```

✅ 2. AVRO 직렬화

📌 개요

항목	설명
형식	이진 포맷 + 스키마
장점	**작고 빠름, 타입 명확, 스키마 진화 가능**
단점	디버깅 어려움, 스키마 관리 필요, JSON보다 복잡

📌 의존성 (Confluent Avro 사용 시)

```
implementation 'io.confluent:kafka-avro-serializer:7.5.0'
implementation 'org.apache.avro:avro:1.11.1'
```

📌 application.yml 설정 (Schema Registry 사용 가정)

```
spring.kafka:
  producer:
    value-serializer: io.confluent.kafka.serializers.KafkaAvroSerializer
  consumer:
    value-deserializer: io.confluent.kafka.serializers.KafkaAvroDeserializer
  properties:
    schema.registry.url: http://localhost:8081
    specific.avro.reader: true
```

📌 Avro 스키마 정의 (user_event.avsc)

```
{
  "type": "record",
  "name": "UserEvent",
  "namespace": "com.example",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "name", "type": "string"}
  ]
}
```

→ `avro-maven-plugin`을 사용해 Java 클래스 자동 생성 가능

📌 메시지 전송

```
UserEvent event = UserEvent.newBuilder()
    .setId(1L)
    .setName("kim")
    .build();

kafkaTemplate.send("user-avro-topic", event);
```

📌 메시지 수신

```
@KafkaListener(topics = "user-avro-topic")
public void listen(UserEvent event) {
    System.out.println("AVRO 이벤트 수신: " + event.getName());
}
```

✅ 3. JSON vs AVRO 비교

항목	JSON	AVRO
형식	텍스트	바이너리
가독성	✅ 높음	❌ 낮음
크기	비교적 큼	매우 작음
속도	느림	빠름
스키마 필요	❌ 없음	✅ 필요
타입 안전성	약함	강함
버전 관리	불편	스키마 진화로 유연함
추천 사용처	디버깅, 내부 API	대용량 로그, MSA 이벤트

✅ 4. Spring 실무 전략

상황	권장 직렬화
내부 서비스, 빠른 개발	JSON 직렬화 (`JsonSerializer`)
대용량 스트리밍, 로그 수집	AVRO (`KafkaAvroSerializer` + Schema Registry)
외부 연동 API	JSON (범용성)
이벤트 스토어	AVRO 또는 Protobuf

✅ 5. 테스트 도구 및 팁

도구	설명
`KafkaTemplate`	메시지 전송
`@KafkaListener`	메시지 수신
Schema Registry	AVRO 스키마 등록/진화
Confluent UI	토픽 메시지 확인
JSON 디코더	`jq`, Postman
AVRO 디코더	`avro-tools`, `kcat`, IntelliJ Plugin

✅ 결론 요약

항목	설명
JSON	간단하고 쉬움. 사람이 읽을 수 있음. 성능은 낮음
AVRO	빠르고 작고 타입 안전. 초기 학습 필요, 스키마 관리 필요
Spring 연동	Kafka: `JsonSerializer`, `KafkaAvroSerializer` 설정으로 처리
실무 전략	디버깅/간단 구조는 JSON, 고성능/대규모 시스템은 AVRO