15. 비동기 처리 및 이벤트 아키텍처

비동기 처리 (`@Async`)

🧩 1. `@Async`란?

> Spring에서 **메서드를 별도의 스레드에서 비동기로 실행**하기 위해 사용하는 어노테이션

- 메서드 호출이 **즉시 반환되고**,
- 내부 로직은 별도의 스레드에서 실행됨
- 병렬 처리, 대기시간 감소, 응답 지연 최소화 등에 유용

⚙️ 2. 사용 전 설정

✅ 비동기 기능 활성화

```
@SpringBootApplication
@EnableAsync
public class MyApplication { ... }
```

> `@EnableAsync`는 반드시 루트 설정 클래스 또는 설정용 `@Configuration` 클래스에 선언해야 함

📘 3. 기본 사용 예

```
@Service
public class NotificationService {

    @Async
    public void sendEmail(String email) {
        // 비동기 메서드
        System.out.println("📧 이메일 발송 중...: " + email);
        // 예: 메일 서버 연결, 대기
    }
}
```

```
notificationService.sendEmail("user@example.com");
// 호출 즉시 반환됨 (실행은 백그라운드 스레드)
```

🔄 4. 반환값이 있는 경우: `Future`, `CompletableFuture`

✅ 비동기 결과 받기

```
@Async
public CompletableFuture<String> findUserName(Long id) {
    // 외부 API 호출 등
    return CompletableFuture.completedFuture("Jeongseok");
}
```

```
CompletableFuture<String> nameFuture = userService.findUserName(1L);
String name = nameFuture.get();  // 비동기 결과 대기
```

> 단, `.get()`을 호출하면 **블로킹 발생** → UI 쓰레드나 Controller에서는 사용 주의

🧨 5. 예외 처리

비동기 메서드에서 예외가 발생하면,

- `void` 반환 → 로그만 찍히고 외부로 전달되지 않음
- `CompletableFuture` 반환 → `.get()` 호출 시 예외 발생

```
@Async
public CompletableFuture<String> failAsync() {
    throw new RuntimeException("예외 발생!");
}
```

```
try {
    future.get();
} catch (ExecutionException e) {
    System.out.println("❗ 비동기 예외: " + e.getCause().getMessage());
}
```

🛠️ 6. 커스텀 Executor 설정

기본은 `SimpleAsyncTaskExecutor` (ThreadPool 없음)

✅ 설정 클래스

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "asyncExecutor")
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);
```

```
        executor.setMaxPoolSize(8);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("async-thread-");
        executor.initialize();
        return executor;
    }
}
```

✅ 특정 Executor 지정

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

🔐 7. 실무에서 자주 쓰는 예시

사용 사례	설명
이메일, 알림 발송	대기 시간 없는 처리
로그 비동기 저장	성능 저하 방지
외부 API 병렬 호출	대기 시간 최적화
파일 저장, 처리	I/O 병목 해소
캐시 프리로딩	서버 시작 후 백그라운드 작업

❗ 8. 주의사항

주의 항목	설명
자기 자신 호출 금지	같은 클래스 내에서 `@Async` 메서드 호출 시 동작 ❌ (프록시 적용 안 됨)
트랜잭션 분리	`@Async` 는 별도 스레드 → 트랜잭션 범위 벗어남
빈 등록 필수	`@Component` 또는 `@Service` 등록된 클래스에서만 적용 가능
동기와 병행 시 주의	`.get()` 을 부주의하게 호출하면 전체 블로킹 유발

✅ 마무리 요약표

항목	설명
`@Async`	메서드 비동기 실행
`@EnableAsync`	설정 클래스에서 활성화 필요
반환값	`void`, `Future`, `CompletableFuture<T>`

항목	설명
예외 처리	`Future.get()` 시 `ExecutionException` 발생
실행 환경	기본 Executor 또는 커스텀 ThreadPool 사용 가능
실무 활용	이메일, SMS, 외부 API, 대용량 처리 등
단점	상태 공유 어려움, 예외 전파 어려움, 트랜잭션 단절 주의

Executor 설정

🧩 1. Executor란?

> **Java에서 비동기 작업을 실행할 스레드 풀(Thread Pool)을 관리하는 인터페이스**

Spring에서는 주로 `ThreadPoolTaskExecutor`를 구현체로 사용하며,
`@Async`, `@Scheduled`, `CompletableFuture`, `@EventListener` 등의 백그라운드 작업에 활용됨.

🛠️ 2. 기본 구성 예시 (`@EnableAsync`와 함께)

```
@Configuration
@EnableAsync
public class AsyncExecutorConfig {

    @Bean(name = "asyncExecutor")
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);                    // 기본 스레드 개수
        executor.setMaxPoolSize(8);                     // 최대 스레드 개수
        executor.setQueueCapacity(100);                 // 큐에 대기할 작업 수
        executor.setKeepAliveSeconds(60);               // 스레드 유휴 시간
        executor.setThreadNamePrefix("async-");         // 디버깅용 이름
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy()); // 꽉 찼을 때 정책
        executor.initialize();
        return executor;
    }
}
```

📘 3. 주요 파라미터 설명

속성	설명
`corePoolSize`	항상 유지되는 최소 스레드 수
`maxPoolSize`	스레드 풀의 최대 크기 (큐가 꽉 찼을 때까지 증가 가능)
`queueCapacity`	대기 중인 작업을 담는 큐 크기 (LinkedBlockingQueue)

속성	설명
`keepAliveSeconds`	초과 스레드가 idle 상태일 때 제거까지 대기 시간
`threadNamePrefix`	로그 추적 등을 위한 스레드 이름 접두어
`rejectedExecutionHandler`	최대치 초과 시 작업을 어떻게 처리할지 결정 (`AbortPolicy`, `CallerRunsPolicy` 등)

⚡ 4. `@Async`와 연동

```
@Async("asyncExecutor")
public void sendNotification(String msg) {
    // async-thread-1 같은 이름으로 실행됨
}
```

✅ Executor 빈의 이름을 `@Async("이름")`으로 지정해야 해당 풀에서 실행됨
❗ 지정하지 않으면 기본 `SimpleAsyncTaskExecutor` 사용 (비권장)

🔄 5. 여러 Executor 분리 운영 (용도별 분리 전략)

```
@Bean(name = "emailExecutor")
public Executor emailExecutor() { ... }

@Bean(name = "reportExecutor")
public Executor reportExecutor() { ... }
```

> 📌 무거운 작업과 경량 작업의 Executor를 분리해 **성능 격리**하는 것이 좋음

💥 6. Rejection Policy 종류

이름	설명
`AbortPolicy`	예외 던짐 (기본값)
`CallerRunsPolicy`	현재 호출 스레드에서 실행 (속도 ↓)
`DiscardPolicy`	아무 경고 없이 버림
`DiscardOldestPolicy`	가장 오래된 작업 버리고 현재 작업 추가

✅ 실무에서는 대부분 `CallerRunsPolicy` 또는 `AbortPolicy`를 사용

🧠 7. 실무 튜닝 기준

항목	권장 값 (예시)	기준
`corePoolSize`	4~N	CPU core 수 기준 (N = core × 2 + I/O 대기 고려)
`maxPoolSize`	corePoolSize × 2~3	예상 peak 처리량 고려
`queueCapacity`	100~1000	요청량 + 처리 시간에 따라 조정
`keepAliveSeconds`	30~60	비활성화 시 리소스 회수 속도

> 🌐 대량 처리 서비스는 모니터링 기반으로 동적 조정 필요 (ex. Actuator, Prometheus)

🔍 8. ThreadPoolTaskExecutor vs Java Executor

항목	ThreadPoolTaskExecutor	Java `ExecutorService`
Spring 관리	✅ Bean 등록 가능	❌ 수동 생성
`@Async`, `@Scheduled` 연동	✅ 가능	❌ 연동 불가
설정 편의성	✅ 편리	❌ 복잡
모니터링/확장	✅ 지원	❌ 수동 관리 필요

✅ 마무리 요약

항목	내용
`ThreadPoolTaskExecutor`	Spring 기본 Executor 구현체
`@Async` 연동	`@Async("beanName")`으로 스레드 풀 지정
핵심 파라미터	core/max size, queue capacity, rejection policy
고성능 운영 기준	작업 종류별 Executor 분리 + CallerRunsPolicy로 fallback 처리
실무 기준	이메일, 로그, 외부 호출 등 병렬 처리에 활용

이벤트 발행 및 구독

- `ApplicationEvent`, `@EventListener`

🧩 1. Spring 이벤트 시스템이란?

> 객체 간 강한 의존 없이 **비동기적으로 메시지(이벤트)를 전달**하는
> **Observer 패턴 기반**의 구조

✅ 핵심 목적: **느슨한 결합, 비동기 흐름, 트랜잭션 이후 처리**

⚙️ 2. 구성 흐름

```
[Event 정의] → [Event 발행 (Publisher)] → [Event 리스너 (@EventListener)]
```

- 이벤트 객체는 DTO처럼 정의
- 발행자는 `ApplicationEventPublisher`를 통해 전송
- 리스너는 `@EventListener` 또는 `implements ApplicationListener` 방식으로 처리

🛠️ 3. 이벤트 클래스 정의

✅ 방식 1: POJO 이벤트 객체

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

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

    // Getter
}
```

📌 4. 이벤트 발행 (Publisher)

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final ApplicationEventPublisher eventPublisher;

    public void registerUser(User user) {
        userRepository.save(user);

        // 이벤트 발행
        eventPublisher.publishEvent(new UserRegisteredEvent(user.getId(),
user.getEmail()));
    }
}
```

✅ 이 시점에서 리스너가 감지 → 실행

👂 5. 이벤트 리스너 (`@EventListener`)

```
@Component
public class WelcomeEmailListener {

    @EventListener
    public void handle(UserRegisteredEvent event) {
        emailService.sendWelcome(event.getEmail());
    }
}
```

✅ 메서드 이름은 자유
✅ 파라미터 타입만 일치하면 자동 매핑됨

🔁 6. 비동기 이벤트 리스너 (`@Async` + `@EnableAsync`)

```
@Async
@EventListener
public void handleAsync(UserRegisteredEvent event) {
    log.info("📧 메일 발송 시작: {}", event.getEmail());
    ...
}
```

> 비동기 이벤트를 사용하려면 `@EnableAsync` 필요하고,
> `@Async`가 붙은 리스너는 **별도 스레드에서 실행됨**

🔄 7. 트랜잭션 이후 이벤트 처리

일반 `@EventListener`는 트랜잭션 커밋 **이전에도** 실행됨.

✅ 트랜잭션 커밋 이후 실행하려면?

```
@Component
public class EventHandler {

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void afterCommit(UserRegisteredEvent event) {
        // DB 트랜잭션이 커밋된 이후에 실행
    }
}
```

> `BEFORE_COMMIT`, `AFTER_COMMIT`, `AFTER_ROLLBACK`, `AFTER_COMPLETION` 등 선택 가능

🧠 8. 실무에서의 이벤트 활용 예시

이벤트	리스너 역할
회원 가입 완료 (`UserRegisteredEvent`)	이메일 발송, Slack 알림
주문 생성 (`OrderCreatedEvent`)	적립금 적립, 배송 예약, 통계 기록
결제 실패 (`PaymentFailedEvent`)	관리자 알림, 자동 취소 예약
게시글 작성 (`PostCreatedEvent`)	검색 인덱스 반영, 추천 시스템 학습

➡️ **도메인 로직과 부가처리를 분리**할 수 있어 유지보수성이 대폭 상승함

❗ 9. 주의사항

항목	주의 설명
트랜잭션 경계	일반 `@EventListener`는 **트랜잭션 내부에서 실행**됨
비동기 예외	`@Async` 리스너의 예외는 무시됨 → 로깅 필요
실행 순서 제어	`@Order(n)` 어노테이션으로 순서 조절 가능
이벤트 순환	이벤트 A → B → A 무한 루프 조심

✅ 마무리 요약

항목	설명
이벤트 정의	POJO 객체로 생성 (`UserRegisteredEvent`)
발행	`ApplicationEventPublisher.publishEvent()`
리스너	`@EventListener`, `@TransactionalEventListener`
비동기 실행	`@Async` + `@EnableAsync`
트랜잭션 이후	`@TransactionalEventListener(phase = AFTER_COMMIT)`
실무 장점	도메인 로직과 부가 로직 분리, 느슨한 결합, 확장성 우수

도메인 이벤트 패턴

🧩 1. 도메인 이벤트란?

> 도메인 모델 내부에서 발생한 **의미 있는 상태 변화**를
> **외부에 알리는 메시지 객체**

📌 예시:

- "회원이 등록되었다" → `UserRegisteredEvent`
- "주문이 결제되었다" → `OrderPaidEvent`
- "상품의 재고가 부족해졌다" → `OutOfStockEvent`

✅ 이벤트는 도메인 내부의 **불변적 사실(Fact)**
✅ 이벤트를 중심으로 **도메인 간 느슨한 연결** 가능

🎯 2. 도메인 이벤트 vs Application 이벤트 차이

항목	도메인 이벤트	ApplicationEvent (Spring)
발행 위치	Entity, Aggregate 내부	Service, Application Layer
의미	도메인 상태 변화 그 자체	시스템 메시지 전파 수단
객체 성격	비즈니스 중심	기술적 중심
예시	`OrderPaidEvent`	`EmailSendRequestedEvent`
단방향 설계	✅ 필수	✅ 선택

> Spring의 `ApplicationEventPublisher`는 도메인 이벤트 전달 수단 중 하나일 뿐

🧱 3. 구조: 어디에서, 어떻게 발행하나?

```
Entity (Aggregate Root)
  ㄴ 상태 변경 메서드 (ex. pay())
        ㄴ 내부에서 도메인 이벤트 생성
              ↓
Application Layer 또는 Event Handler가 수신
```

📘 4. 예제 – 주문 결제 후 배송 예약

✅ 1) 도메인 이벤트 정의

```java
public class OrderPaidEvent {
    private final Long orderId;
    private final Long userId;

    public OrderPaidEvent(Long orderId, Long userId) {
        this.orderId = orderId;
        this.userId = userId;
    }
}
```

✅ 2) Aggregate Root에서 이벤트 생성

```java
@Entity
public class Order {

    @Transient
    private final List<Object> domainEvents = new ArrayList<>();

    public void pay() {
        this.status = OrderStatus.PAID;
        domainEvents.add(new OrderPaidEvent(this.id, this.user.getId()));
    }

    public List<Object> getDomainEvents() {
        return domainEvents;
    }

    public void clearEvents() {
        domainEvents.clear();
    }
}
```

✅ 3) Application Layer에서 이벤트 발행

```
@Transactional
public void completePayment(Long orderId) {
    Order order = orderRepository.findById(orderId).orElseThrow();
    order.pay();

    orderRepository.save(order);

    // 도메인 이벤트 수동 발행
    order.getDomainEvents().forEach(eventPublisher::publishEvent);
    order.clearEvents();
}
```

> ☝️ 핵심: 이벤트는 **도메인 객체가 만들고**, 발행은 **Application Layer가 담당**

✅ 4) EventListener 처리

```
@Component
public class ShippingReservationHandler {

    @EventListener
    public void on(OrderPaidEvent event) {
        shippingService.reserveShipment(event.getOrderId());
    }
}
```

🔄 5. 트랜잭션 이후 실행하려면?

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void on(OrderPaidEvent event) {
    ...
}
```

➡️ 트랜잭션 커밋 이후에만 배송 예약됨 → **DB 롤백 방지 가능**

⚙️ 6. Spring에서의 통합 흐름

```
[Entity]
  상태변경 → 이벤트 객체 생성
    ↓
[ApplicationService]
  저장 + publishEvent() 호출
    ↓
@EventListener / @TransactionalEventListener 수신
    ↓
비동기 처리, API 호출, 메시지 전송 등
```

💡 7. 실무 설계 장점

항목	설명
느슨한 결합	주문 도메인 ↔ 배송/포인트/이메일 시스템 분리
확장성	이벤트 리스너만 추가하면 기능 추가 가능
테스트 용이	도메인 단위 테스트 시 이벤트 생성 여부 검증 가능
단일 책임	도메인은 상태 변화만, 후처리는 다른 모듈이 담당

✅ 마무리 요약표

항목	내용
이벤트 위치	Entity/Aggregate 내부에서 생성
발행 책임	Application Layer가 `publishEvent()` 수행
처리 위치	`@EventListener`, `@TransactionalEventListener`
트랜잭션 경계	AFTER_COMMIT 설정 필수 (권장)
장점	결합도↓, 확장성↑, 테스트 가능, 분산 시스템 연계 용이
실무 적용	사용자 가입 → 이메일 발송, 포인트 적립, 슬랙 알림 등

트랜잭션 이벤트 처리

🧩 1. 개념

> 트랜잭션 커밋 이후에 이벤트를 안전하게 실행하고 싶을 때 사용하는
> Spring의 이벤트 리스너 어노테이션

✅ `@EventListener`는 **트랜잭션 중에도 실행**되지만,
✅ `@TransactionalEventListener`는 **트랜잭션 상태를 감지**해서 실행 시점을 선택할 수 있음

🔄 2. 실행 시점 (`TransactionPhase`)

옵션	실행 시점	설명
`AFTER_COMMIT` ✅	트랜잭션 커밋 직후	**실무 기본값**, 데이터 반영 후
`AFTER_ROLLBACK`	트랜잭션 롤백 시	오류 감지 후 후처리
`AFTER_COMPLETION`	커밋 또는 롤백 후	무조건 마지막
`BEFORE_COMMIT`	커밋 직전	롤백 발생 시 함께 롤백됨

📘 3. 기본 사용 예시

✅ 1) 이벤트 클래스

```
public class OrderCreatedEvent {
    private final Long orderId;

    public OrderCreatedEvent(Long orderId) {
        this.orderId = orderId;
    }
}
```

✅ 2) 이벤트 발행

```
@Service
@RequiredArgsConstructor
public class OrderService {

    private final ApplicationEventPublisher eventPublisher;

    @Transactional
    public void createOrder(Order order) {
        orderRepository.save(order);
        eventPublisher.publishEvent(new OrderCreatedEvent(order.getId()));
    }
}
```

✅ 3) 트랜잭션 이벤트 리스너

```
@Component
public class OrderEventHandler {

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void handleOrderCreated(OrderCreatedEvent event) {
        System.out.println("✅ 주문 이벤트 처리됨 (commit 이후): " + event.getOrderId());
    }
}
```

> ☝️ DB에 주문 저장이 **성공적으로 커밋된 이후**에만 실행됨

🧠 4. 왜 중요한가?

상황	이유
이메일 발송	주문 저장 실패 시 발송 X

상황	이유
포인트 적립	롤백되면 적립도 취소되어야 함
Slack 알림	커밋된 상태에서만 알림 보내야 신뢰성 보장
Kafka 메시지 발행	커밋 전 발행하면 메시지와 DB가 불일치 가능

✅ 반드시 **AFTER_COMMIT**에서 처리해야 **이벤트 > 외부 시스템의 데이터 정합성 유지** 가능

🧨 5. 예외 처리 전략

- `@TransactionalEventListener` 메서드에서 예외 발생 시,
  트랜잭션은 이미 커밋된 상태라 롤백 불가

➡️ **로그 남기기**, **Retry**, **Dead-letter 처리** 등을 따로 구현해야 함

```
@SneakyThrows
@TransactionalEventListener(phase = AFTER_COMMIT)
public void sendNotification(OrderCreatedEvent event) {
    try {
        slackNotifier.send("주문 완료: " + event.getOrderId());
    } catch (Exception ex) {
        log.error("❗ 슬랙 알림 실패: {}", ex.getMessage());
        // 추후 재시도 큐에 적재할 수 있음
    }
}
```

⚙️ 6. 비동기 처리와 함께 사용 가능 (`@Async`)

```
@Async
@TransactionalEventListener(phase = AFTER_COMMIT)
public void processAsync(OrderCreatedEvent event) {
    log.info("비동기 후처리 실행: {}", event.getOrderId());
}
```

✅ 비동기 + 커밋 이후 실행됨 → 외부 API나 I/O 작업 적합
❗ 비동기 예외는 로그만 출력되고 전파되지 않음 → 반드시 예외 핸들링 필수

🧱 7. 아키텍처에서의 위치

```
[Domain] - 도메인 이벤트 생성
   ↓
[Application Layer] - eventPublisher.publishEvent()
   ↓
@TransactionalEventListener
   ↓
비동기 실행, 외부 API, 메시지 전송, 통계 기록 등
```

🧩 8. 실무 설계 팁

항목	권장 전략
도메인 이벤트 발행 위치	Entity or Service 내부에서 생성
발행 책임	Application Layer (`publishEvent`)에서 명시적으로 수행
Listener	`@TransactionalEventListener(AFTER_COMMIT)` 로 후처리
예외 처리	try-catch 후 재시도 큐 또는 로깅
리스너 격리	`@Async` 로 병렬성 확보 + timeout 대비

✅ 마무리 요약표

항목	설명
`@TransactionalEventListener`	트랜잭션 상태 감지 후 이벤트 처리
기본 phase	`AFTER_COMMIT`
예외 처리	트랜잭션과 별개 → 자체 try-catch 필수
비동기 지원	`@Async` 함께 사용 가능
실무 목적	이벤트 처리 순서 보장 + DB 정합성 유지 + 외부 시스템과 안정 연동