5. 서비스 계층 (Service Layer)

`@Service` 클래스의 역할

🔍 1. `@Service`란?

`@Service`는 Spring Framework에서 **비즈니스 로직을 처리하는 계층(=서비스 계층)**에 붙이는 컴포넌트 어노테이션이다.

> 내부적으로는 `@Component`의 특수화(Stereotype)이며,
> **비즈니스 트랜잭션과 도메인 조합의 중심**으로 동작한다.

🧭 2. 서비스 계층(Service Layer)의 목적

목표	설명
📦 비즈니스 흐름 구현	도메인 객체, Repository, 외부 시스템 등을 조합해 복잡한 유스케이스 실행
🔁 트랜잭션 경계 설정	`@Transactional`로 DB 작업을 묶어서 실행/롤백
🧠 도메인 로직 호출	도메인 객체가 정의한 핵심 로직을 **위임**하여 실행
🔄 외부 시스템 연계	결제 API, 이메일 발송, Kafka 전송 등 도메인 외부 요소 호출 포함 가능
🧪 단위 테스트 대상	로직이 집중되므로 테스트 중심 계층이기도 함

⚙️ 3. 전형적인 역할 요약

기능	역할
💡 유스케이스 조립	여러 도메인 객체/Repo의 순서 정의
🔐 입력 검증/조건 판단	특정 조건에 따라 흐름 분기 (도메인 이전 단계의 조건 판단)
🔁 트랜잭션 설정	`@Transactional` 사용
🧱 도메인 호출	`user.createAccount()`, `order.markAsPaid()`
☁️ 외부 호출	`paymentGateway.charge(...)`, `emailSender.send(...)`

🧪 4. 실전 예시

```
@Service
public class OrderService {

    private final OrderRepository orderRepository;
    private final InventoryRepository inventoryRepository;
    private final PaymentGateway paymentGateway;
```

```
7
8       @Transactional
9       public void placeOrder(Long userId, Long itemId, int quantity, PaymentInfo
    paymentInfo) {
10          Inventory inventory = inventoryRepository.findByItemId(itemId);
11          inventory.decrease(quantity);
12
13          paymentGateway.charge(paymentInfo);
14
15          Order order = new Order(userId, itemId, quantity);
16          order.markAsPaid();
17
18          orderRepository.save(order);
19      }
20  }
```

🔐 5. `@Service`의 위치와 어노테이션

어노테이션	설명
`@Service`	Spring 컨테이너에 등록. 서비스 역할 명시
`@Transactional`	트랜잭션 경계 설정. 반드시 서비스 계층에 선언할 것
`@RequiredArgsConstructor`	생성자 주입을 편하게 설정 (Lombok 사용 시)

📚 6. 안티패턴 피하기

안티패턴	설명
❌ Fat Service	모든 로직이 서비스에 몰려있고 도메인은 빈 껍데기 → 비즈니스 규칙은 도메인으로 분리할 것
❌ 도메인 무시	서비스가 엔티티의 상태를 직접 변경 (`order.setStatus(...)`) ← 반드시 도메인 메서드를 통해 변경
❌ 무의미한 패싯(Facade)화	단순 CRUD만 하는 `UserService.save()`, `findById()`는 존재 이유가 없음. Controller → Repository로 직접 가는 게 낫다

✅ 도메인 주도 설계(DDD) 관점 요약

계층	역할
Controller	DTO 매핑, 요청 수신, Command 변환
Service (`@Service`)	유스케이스 흐름 정의, 도메인/외부 조합, 트랜잭션 조율
Domain	핵심 상태 변경 및 비즈니스 규칙 정의

🧠 언제 `@Service`를 만들고 언제 만들지 말아야 할까?

만들 때	안 만들 때
유스케이스가 복잡함	단순 CRUD만 필요
도메인 객체 여러 개를 조합	한 개 엔티티에 대해 단순 조회만 할 때
외부 시스템 호출 포함	단순 조회는 `@Repository` 직접 호출이 더 적합
트랜잭션 범위가 있음	로직이 없고 단순 래핑만 하면 의미 없음

✅ 요약 정리표

항목	설명
정의	비즈니스 유스케이스를 실행하는 계층
핵심 책임	트랜잭션, 도메인 호출, 외부 시스템 연동
어노테이션	`@Service`, `@Transactional`
위치	Application Layer or Service Layer
SRP 기준	흐름 조정에 집중, 규칙은 도메인으로 위임
테스트 단위	단위 테스트, mocking 테스트 중심 계층

트랜잭션 범위 지정: `@Transactional`

🧩 1. `@Transactional`이란?

Spring Framework가 제공하는 **선언적 트랜잭션 관리 기능**
→ 메서드 실행을 **트랜잭션 범위로 감싸고**,
→ 예외가 발생하면 **자동 롤백**,
→ 정상 종료되면 **커밋**하는 기능을 수행함.

```
@Transactional
public void process() {
    // 트랜잭션 시작
    ... DB 작업 ...
    // 트랜잭션 커밋 or 롤백
}
```

🎯 2. 기본 동작 방식

항목	기본값
전파 방식 (propagation)	`REQUIRED`
격리 수준 (isolation)	데이터베이스 기본값
롤백 조건 (rollbackFor)	`RuntimeException`, `Error` 계열만
읽기 전용 (readOnly)	`false`

📦 3. 선언 위치와 의미

선언 위치	의미
클래스 전체	모든 public 메서드에 트랜잭션 적용
메서드 개별	해당 메서드에만 트랜잭션 적용 (우선순위 더 높음)

```
@Service
@Transactional
public class OrderService {
    public void placeOrder() {} // 트랜잭션 O
}

@Transactional
public void cancelOrder() {} // 개별 선언 시 우선
```

⚙️ 4. 트랜잭션 전파 속성 (`propagation`)

속성	의미
`REQUIRED` (기본값)	이미 트랜잭션이 있으면 참여, 없으면 새로 시작
`REQUIRES_NEW`	기존 트랜잭션 **일시 중단**, **새 트랜잭션 생성**
`MANDATORY`	반드시 기존 트랜잭션이 있어야 함, 없으면 예외
`NESTED`	내부에 별도 저장점(Savepoint) 만들어 하위 롤백 가능
`NEVER`	트랜잭션이 존재하면 예외
`SUPPORTS`	있으면 참여, 없으면 비트랜잭션으로 수행
`NOT_SUPPORTED`	기존 트랜잭션을 **중지**하고 비트랜잭션 수행

🧪 5. 롤백 전략 (`rollbackFor`, `noRollbackFor`)

✔ 기본 동작

- `RuntimeException`, `Error` → 롤백 O
- `Checked Exception` (`IOException`, `SQLException`) → 롤백 ❌

✔ 예시

```
@Transactional(rollbackFor = Exception.class)
public void saveWithChecked() throws Exception {
    // checked 예외 발생해도 롤백하도록 지정
}
```

🧠 6. 내부 동작 원리: 프록시 기반 AOP

- Spring은 `@Transactional`이 붙은 메서드를 **프록시로 감싸서 트랜잭션 로직을 삽입**함
- 실제 메서드 호출 전후로 `PlatformTransactionManager`가 **트랜잭션 시작/커밋/롤백**을 수행

> 📌 내부 호출(self-invocation)은 트랜잭션 적용 ❌
> 같은 클래스 안에서 `this.methodA()` 호출 → 프록시 우회되므로 트랜잭션 적용 안 됨

📌 7. 실전 주의사항

주의 항목	설명
내부 호출	`this.메서드()`로 같은 클래스 메서드 호출 시 트랜잭션 적용 안 됨 (→ 분리 필요)
private 메서드	프록시 기반이라 `private` 메서드에 `@Transactional` 붙여도 효과 없음
readOnly	`@Transactional(readOnly = true)` → select 전용 트랜잭션, flush 최적화 가능
예외 누락	예외를 잡고 `.printStackTrace()`만 하면 롤백 안 됨 (→ 반드시 throw 해야 함)

📚 8. 예시 정리

일반 사용

```
@Service
public class UserService {

    @Transactional
    public void registerUser() {
        userRepository.save(...);
        emailSender.send(...);
    }
}
```

읽기 전용

```
@Transactional(readOnly = true)
public User getUser(Long id) {
    return userRepository.findById(id).orElseThrow();
}
```

새 트랜잭션 분리

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void logAudit() {
    auditLogRepository.save(...); // 본 트랜잭션과 분리
}
```

✅ 트랜잭션 동작 흐름 요약

```
1. 프록시 객체가 메서드 호출을 가로챔
2. 트랜잭션 시작 (TransactionManager)
3. 실제 비즈니스 로직 수행
4. 예외 발생 여부 확인
   - 예외 없음 → Commit
   - 예외 있음 → Rollback
```

✅ 마무리 요약표

항목	설명
핵심 기능	트랜잭션 시작/커밋/롤백 자동화
기본 전파	REQUIRED (트랜잭션 있으면 참여, 없으면 생성)
롤백 대상	RuntimeException, Error
비적용 조건	내부 호출, private 메서드
트랜잭션 분리	REQUIRES_NEW 사용
읽기 전용 최적화	`readOnly = true`

다수의 Repository 및 외부 API 통합 처리

🧩 1. 상황 예시

> 다음과 같은 복합 유스케이스를 생각해보자:

```
"회원이 주문을 생성하면,
- 로컬 DB에 주문을 저장하고
- 결제 서비스(API)로 결제 요청을 보내며,
- 포인트 서비스를 통해 회원 포인트를 차감한다."
```

➡️ 이때 필요한 구성:

- `OrderRepository`, `UserRepository`, `PointRepository` (JPA)
- 외부 API 클라이언트: `PaymentApiClient`, `PointApiClient` (RestTemplate/WebClient/Feign)
- **단일 트랜잭션처럼 묶여야 하고**, 실패 시 롤백 or 보상 필요

📦 2. 핵심 구성 계층 구조

```
Controller
  ↓
ApplicationService (비즈니스 유스케이스)
  ├─ Repository (JPA 기반)
  ├─ External API Client
  ↓
Domain (Entity, Domain Service)
```

🎯 3. Application Service 역할

> 외부 시스템 + 내부 Repository를 조합해 **트랜잭션 흐름을 제어**하고
> **오류 발생 시 전체 실패 or 보상 트랜잭션 처리**를 수행

```
@Transactional
public void placeOrder(PlaceOrderCommand command) {
    User user = userRepository.findById(command.getUserId())
        .orElseThrow(() -> new NotFoundException("사용자 없음"));

    Order order = Order.create(user, command);
    orderRepository.save(order); // 1. 로컬 저장

    paymentClient.requestPayment(order);     // 2. 외부 결제
    pointClient.deduct(user.getId(), 1000);  // 3. 외부 포인트 차감

    // 성공 시 이벤트 발행 등
}
```

🔁 4. 트랜잭션 처리 전략

대상	트랜잭션 처리
JPA Repository	`@Transactional`로 관리

대상	트랜잭션 처리
외부 API 호출	DB 트랜잭션 이후 호출 권장 (`commit` 직전 flush 후)
실패 보상	API 호출 실패 시 **로컬 DB rollback + 보상 API 호출** 필요

> Spring은 JPA 트랜잭션 안에서 외부 HTTP 호출 시 **예외 발생하면 자동 rollback**

📘 5. 외부 API 호출 예시 (WebClient 기반)

```
public class PaymentApiClient {
    private final WebClient webClient;

    public void requestPayment(Order order) {
        PaymentRequest payload = PaymentRequest.of(order);
        PaymentResponse response = webClient.post()
            .uri("/api/pay")
            .bodyValue(payload)
            .retrieve()
            .onStatus(HttpStatus::isError, res -> Mono.error(new
PaymentFailException()))
            .bodyToMono(PaymentResponse.class)
            .block();  // 블로킹 호출
    }
}
```

❌ 6. 실패 케이스 처리

실패 구간	처리 방식
`orderRepository.save()` 실패	`@Transactional` rollback
`paymentClient` 실패	예외 throw → rollback됨
`pointClient` 실패	예외 throw + rollback → 보상 필요 (ex: 취소 결제 요청)
외부 API 일부만 성공한 경우	별도 `CompensationService`를 통해 복구 플로우 구성

🧠 7. 실무 설계 기준

항목	기준
비즈니스 단위 묶음	반드시 Application Service로 통합
로컬 트랜잭션	`@Transactional`로 묶고, 외부 API는 후반부에 호출
실패 예외 관리	커스텀 예외 정의 + `@ControllerAdvice`로 대응

항목	기준
통합 응답 객체	DTO로 통합하여 반환 (`OrderResultResponse`)
성능	외부 API는 `WebClient` + 비동기 가능 (단, 트랜잭션 분리 주의)

📦 8. 응답 DTO 설계 예시

```
public class OrderResultResponse {
    private final Long orderId;
    private final String paymentStatus;
    private final int remainingPoints;

    public static OrderResultResponse of(Order order, PaymentResult payment,
PointBalance point) {
        return new OrderResultResponse(order.getId(), payment.getStatus(),
point.getBalance());
    }
}
```

✅ 9. 예외 핸들링 (`@ControllerAdvice`)

```
@ExceptionHandler(PaymentFailException.class)
public ResponseEntity<ApiResponse<Void>> handlePaymentError(PaymentFailException ex) {
    return ResponseEntity.status(502).body(ApiResponse.error("PAYMENT_FAIL", "결제에 실패
했습니다."));
}
```

✅ 마무리 요약표

항목	설계 기준
트랜잭션 단위	ApplicationService 단위로 묶고, 외부 API는 마지막에 호출
DB와 API 통합	JPA는 `@Transactional` , API는 예외 발생 시 전체 롤백
외부 API 실패 처리	예외 + 로그 + (필요 시) 보상 트랜잭션 구성
응답 객체	DTO로 통합 설계, 클라이언트 해석 가능하게
예외 구조	`CustomException` , `ErrorCode` , `@ControllerAdvice` 조합으로 관리

서비스 계층에서의 유효성 검사, 비즈니스 규칙 적용

🧩 1. 유효성 검사의 계층 구분

계층	유효성 검사 목적
Controller	**입력값 구조**, 형식, 필수값 검사 (`@Valid`, `@NotNull` 등)
Service	**도메인 비즈니스 규칙**, 상태 검증, 존재성 검사 등
Domain (Entity, DomainService)	**불변성**, **상태 전이**, **자기완결 규칙**

🚫 2. 왜 서비스 계층에서도 유효성 검사가 필요한가?

> 컨트롤러에서는 **단순 값 유효성**만 체크함 (null, length, pattern)

하지만 다음과 같은 검사는 서비스 계층에서만 가능함:

- 사용자 존재 여부 확인
- 주문 상태가 `PENDING`인지 확인
- 포인트가 충분한지 검사
- 중복 주문 여부
- 특정 정책(예약 7일 전까지만 가능 등)

✅ 이건 전부 **비즈니스 규칙**, 즉 **서비스 계층의 책임**

📘 3. 예시: 주문 생성 유스케이스

요청 DTO

```
public class OrderRequest {
    @NotNull
    private Long userId;

    @NotNull
    private Long productId;

    @Min(1)
    private int quantity;
}
```

컨트롤러

```
@PostMapping("/orders")
public ResponseEntity<?> placeOrder(@Valid @RequestBody OrderRequest req) {
    orderService.placeOrder(req);
    return ResponseEntity.status(HttpStatus.CREATED).build();
}
```

✅ 4. 서비스 계층에서의 유효성 검사

```
@Transactional
public void placeOrder(OrderRequest req) {
    User user = userRepository.findById(req.getUserId())
        .orElseThrow(() -> new NotFoundException("사용자 없음"));

    Product product = productRepository.findById(req.getProductId())
        .orElseThrow(() -> new NotFoundException("상품 없음"));

    if (product.getStock() < req.getQuantity()) {
        throw new InvalidRequestException("재고가 부족합니다.");
    }

    if (!user.canOrder()) {
        throw new BusinessRuleException("정지된 사용자입니다.");
    }

    Order order = Order.create(user, product, req.getQuantity());
    orderRepository.save(order);
}
```

💣 5. 잘못된 설계 예시

```
// controller에서 모든 유효성 검사 처리
if (!productService.hasStock(productId, quantity)) {
    return ResponseEntity.badRequest().body("재고 부족");
}
```

❌ 문제점:

- 유효성 검사가 여러 컨트롤러에 중복됨
- 서비스 테스트 시 규칙이 빠질 수 있음
- 상태 전이 검사 누락 위험

🎯 6. 유효성 검사 & 예외 구조 패턴

✅ 커스텀 예외 클래스

```
public class BusinessRuleException extends RuntimeException {
    private final String code = "BUSINESS_RULE_VIOLATION";
}
```

✅ 전역 처리

```
@ExceptionHandler(BusinessRuleException.class)
public ResponseEntity<ApiResponse<Void>> handleRule(BusinessRuleException ex) {
    return ResponseEntity.badRequest().body(ApiResponse.error(ex.getCode(),
ex.getMessage()));
}
```

🧠 7. 비즈니스 규칙의 위치 구분

구분	처리 위치	예
단순 값 검증	Controller (`@Valid`)	email 형식, 비어있음 등
DB 존재성, 상태 조건	Service Layer	존재 여부, 중복 검사, 상태 조건 등
엔티티 불변성, 상태 전이	Entity 또는 Domain Service	Order 상태 변경, 정책적 제약 등

🧪 8. 단위 테스트 설계

서비스 계층의 유효성 검사는 반드시 테스트되어야 함:

```
@Test
void 주문요청_재고부족_예외발생() {
    given(product.getStock()).willReturn(0);

    assertThrows(InvalidRequestException.class, () -> {
        orderService.placeOrder(req);
    });
}
```

✅ 마무리 요약표

항목	설명
컨트롤러	단순 구조/값 유효성 (`@Valid`)
서비스	비즈니스 규칙, 상태 조건, 관계 유효성 검사
도메인	상태 전이, 불변성, 핵심 정책
예외	커스텀 예외 + 전역 처리 (`@ControllerAdvice`)
테스트	서비스 계층 규칙은 테스트 필수 (행위 중심)

예: `UserService`, `OrderService`, `PaymentService`

🔹 1. `UserService`

✅ 역할

기능	설명
회원 조회	단건/전체 조회
회원 등록	중복 검사 포함
회원 상태 관리	휴면/탈퇴 처리 등
포인트 관리	적립, 차감, 잔액 조회 등

✅ 주요 메서드 시그니처

```
UserDto getUser(Long id);
void registerUser(UserCreateCommand command);
void deactivateUser(Long id);
void addPoint(Long userId, int amount);
void subtractPoint(Long userId, int amount);
```

✅ 내부 유효성 및 규칙

- 이메일 중복 검사
- 존재하지 않는 사용자 처리
- 정지/탈퇴 사용자는 주문 불가

```
if (user.isDeactivated()) {
    throw new BusinessRuleException("탈퇴한 회원은 사용할 수 없습니다.");
}
```

🔹 2. `OrderService`

✅ 역할

기능	설명
주문 생성	상품 존재 여부 + 재고 검증 포함
주문 취소	상태 조건(배송 전) 검증
주문 목록 조회	사용자별 또는 전체 조회
결제 연계	내부 저장 → 외부 결제 요청

✅ 주요 메서드 시그니처

```
OrderDto placeOrder(OrderRequest command);          // 주문 생성
void cancelOrder(Long orderId);                     // 주문 취소
List<OrderDto> getOrdersByUser(Long userId);        // 사용자별 주문 목록
```

✅ 내부 검증

- 상품 재고 확인 (`product.getStock() < qty`)
- 사용자 존재, 상태 확인
- 주문 상태가 `READY`인 경우에만 취소 허용

```
if (!order.isCancelable()) {
    throw new BusinessRuleException("배송 준비 상태에서는 취소할 수 없습니다.");
}
```

🔹 3. `PaymentService`

✅ 역할

기능	설명
결제 요청	외부 PG사 API 연계
결제 취소	상태/시간 조건 검증
결제 상태 조회	외부 응답 → 내부 상태 반영
결제 결과 저장	결제 내역 로깅

✅ 주요 메서드 시그니처

```
PaymentResult pay(PaymentCommand command);
void cancelPayment(Long paymentId);
PaymentStatus getStatus(Long orderId);
```

✅ 외부 API 연계 (Feign, WebClient)

```
public PaymentResult pay(PaymentCommand command) {
    try {
        PaymentApiResponse response = paymentApiClient.requestPayment(command);
        return PaymentResult.from(response);
    } catch (HttpClientErrorException e) {
        throw new PaymentFailException("결제 실패: " + e.getMessage());
    }
}
```

🔐 트랜잭션 처리 정리

서비스	트랜잭션 여부	예외 시 처리
`UserService`	등록, 상태 변경 시 `@Transactional`	롤백
`OrderService`	주문 생성, 취소는 반드시 `@Transactional`	실패 시 전체 롤백
`PaymentService`	DB 저장은 `@Transactional`, 외부 API는 트랜잭션 이후 호출 권장	예외 시 보상 트랜잭션 필요

⚠️ 예외 설계 예시

```
public class UserNotFoundException extends RuntimeException { ... }
public class OutOfStockException extends RuntimeException { ... }
public class PaymentFailException extends RuntimeException { ... }
```

✅ 마무리 구조 요약

서비스	주요 기능	트랜잭션 여부	외부 연계	유효성 검사
UserService	등록, 탈퇴, 포인트	✅	❌	사용자 상태, 중복
OrderService	주문 생성, 취소	✅	☑️ (간접적으로 PaymentService 연동)	상품, 재고, 사용자
PaymentService	결제, 취소, 상태	부분 ✅	✅ PG API	결제 요청 가능 여부, 상태 확인