삼삼오토

三품관리

현대오토에버 모빌리티 SW스쿨 웹/앱 3조

팀장 - 추창우
팀원 - 김태민
성현주
양지선
이종진
채상윤

Contents

01 프로젝트 개요 - 프로젝트 주제

주제 소개

다중 지점(Multi-Branch) 운영 및 생산/물류(WMS) 관리를 위한 통합 ERP(전사적 자원 관리) 시스템 구축

프로젝트 목표

프로세스 통합

구매, 생산, 창고, 판매, 인사 등 기업의 핵심 업무를 하나의 시스템으로 연결합니다.

데이터 일관성 확보

일관된 기준정보(Master Data)를 기반으로 전 모듈이 동작하여 데이터의 중복과 오류를 제거합니다.

업무 효율화

특히 창고(WMS) 및 생산 현장의 프로세스를 표준화하고, 재주문점(ROP) 등을 통해 자동화 기반을 마련하여 업무 효율을 극대화합니다.

실시간 가시성 확보

본사와 지점 간, 또는 각 부서 간의 재고 및 업무 현황을 실시간으로 파악하여 정확한 의사결정을 지원합니다.

02 팀 구성 및 역할

추창우

프론트엔드 개발
ERP대장
추하하하..

김태민

백엔드 개발
공장장
구매 관리

성현주

백엔드 개발
창고 관리
판매 관리

양지선

백엔드 개발
대리점장
기준 정보 관리

이종진

백엔드 개발
인증/인가
인사 관리

채상윤

모바일 개발
Android
iOS

03 기획 및 설계 - 요구 사항 정의

MDM

- 모든 업무 프로세스의 데이터 기반
- 품목(Item), 자재명세서(BOM), 거래처(Partner), 작업장(WorkCenter) 등 시스템 전체가 공유하는 '데이터 표준'을 정의하고 관리

판매

- 고객사의 '판매 주문'을 접수하고 관리한다.
- 접수된 주문은 '출고(WMS)' 프로세스 또는 '생산(Production)' 프로세스의 시작점이 되어야 한다.

WMS

- 모든 물적 자원의 흐름(입/출고)과 보관을 담당하는 핵심 현장 모듈이다.
- 입고(Receiving), 적치(Stocking), 재고(Inventory) 관리, 출고(Shipping) 및 재주문점(ROP) 설정을 포함해야 한다.

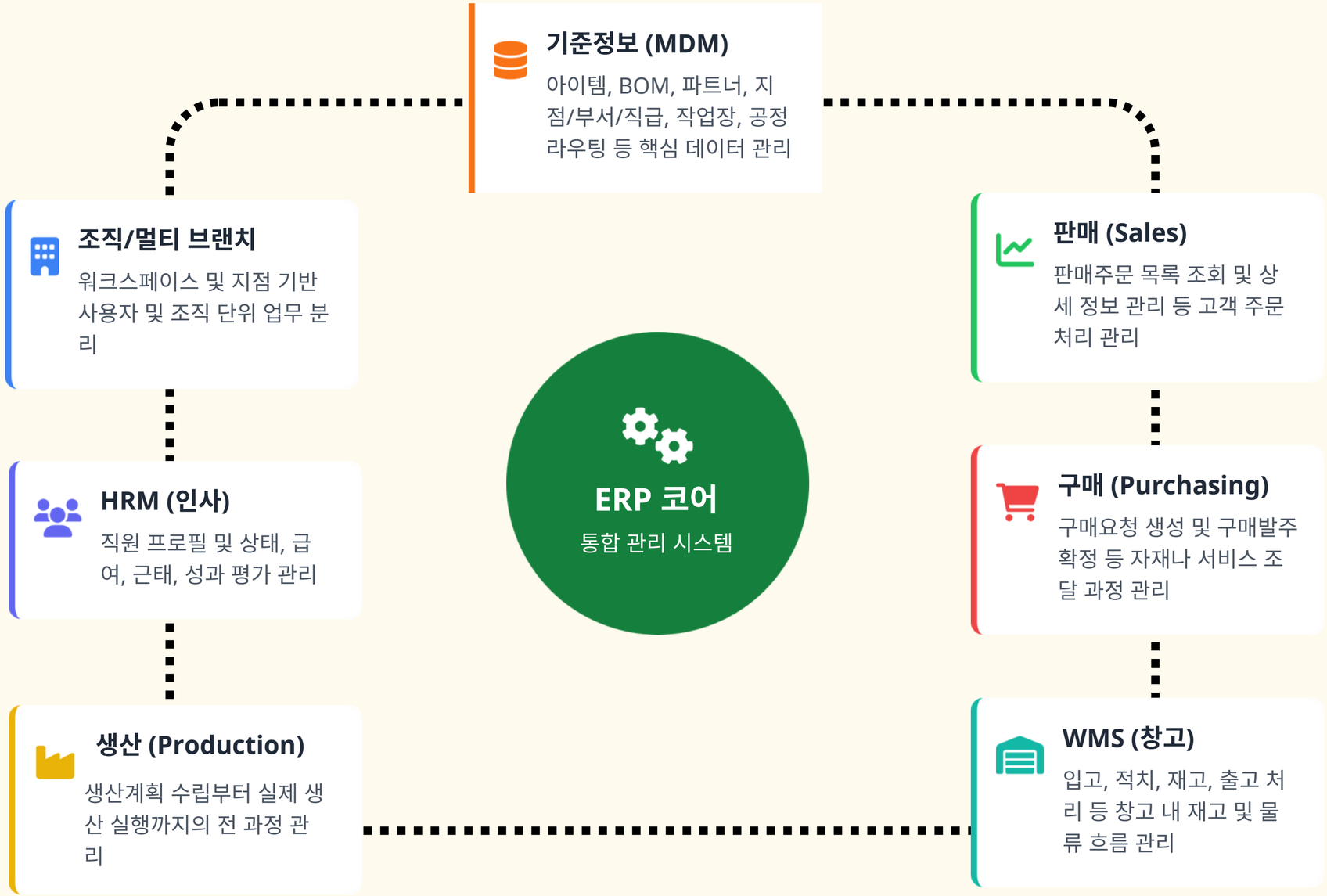
생산

- 판매 계획 또는 재고 상황에 따라 실제 제품을 만들어내는 모듈이다.
- '주 생산 계획(MPS)' 수립, '자재 소요량(MRP)' 계산 ,'작업 지시(MES)' 발행, '공정 라우팅' 이행 및 실적 집계 기능을 정의한다.

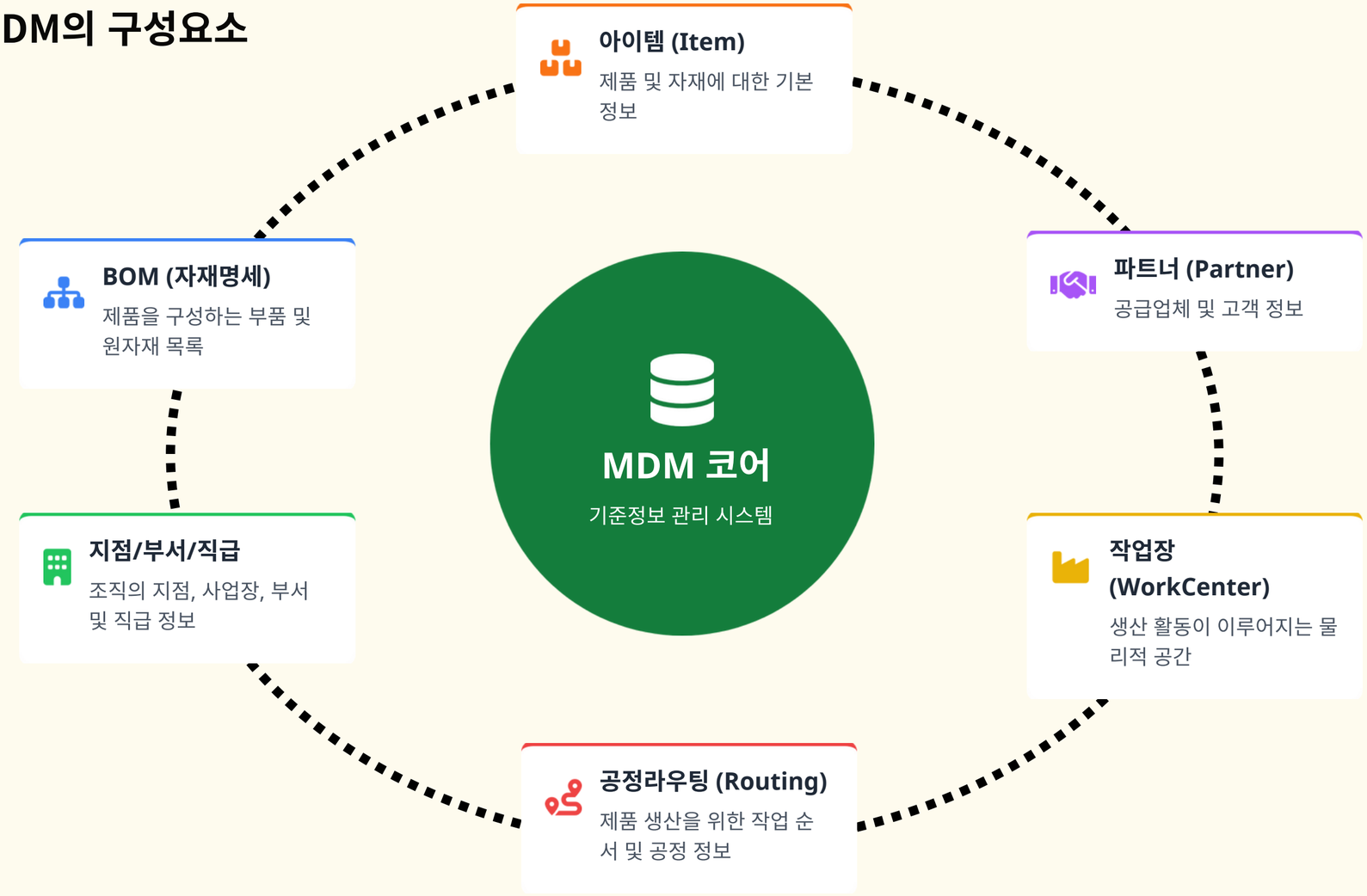
구매

- 생산에 필요한 원자재를 조달하는 역할을 수행

03 기획 및 설계 - 구성요소 정의



03 기획 및 설계 - MDM의 구성요소

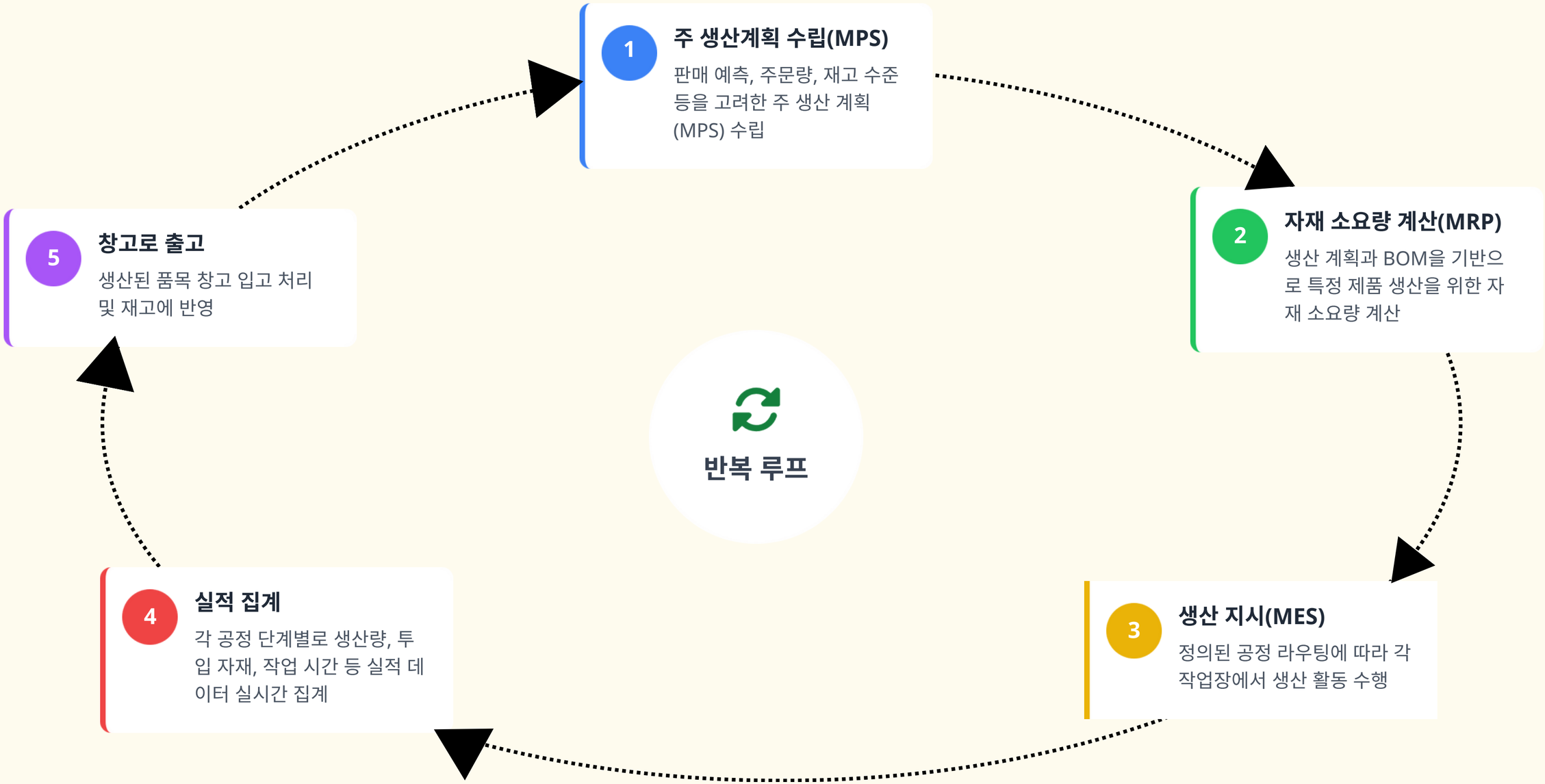


필요한 이유

- **MDM(Master Data Mangement)은 ERP 시스템 운영의 근간이 되는 핵심 데이터를 관리하며, 모든 모듈에서 참조 무결성을 유지하는 데 필수적입니다.**
- **통합된 시스템 내에서 각 모듈은 이 데이터를 공유하고 상호 연동되어 효율적인 업무 처리를 지원합니다**
- **저희는 이것을 저희 삼삼오토 ERP의 '유비쿼터스 언어'로 정의했습니다.**

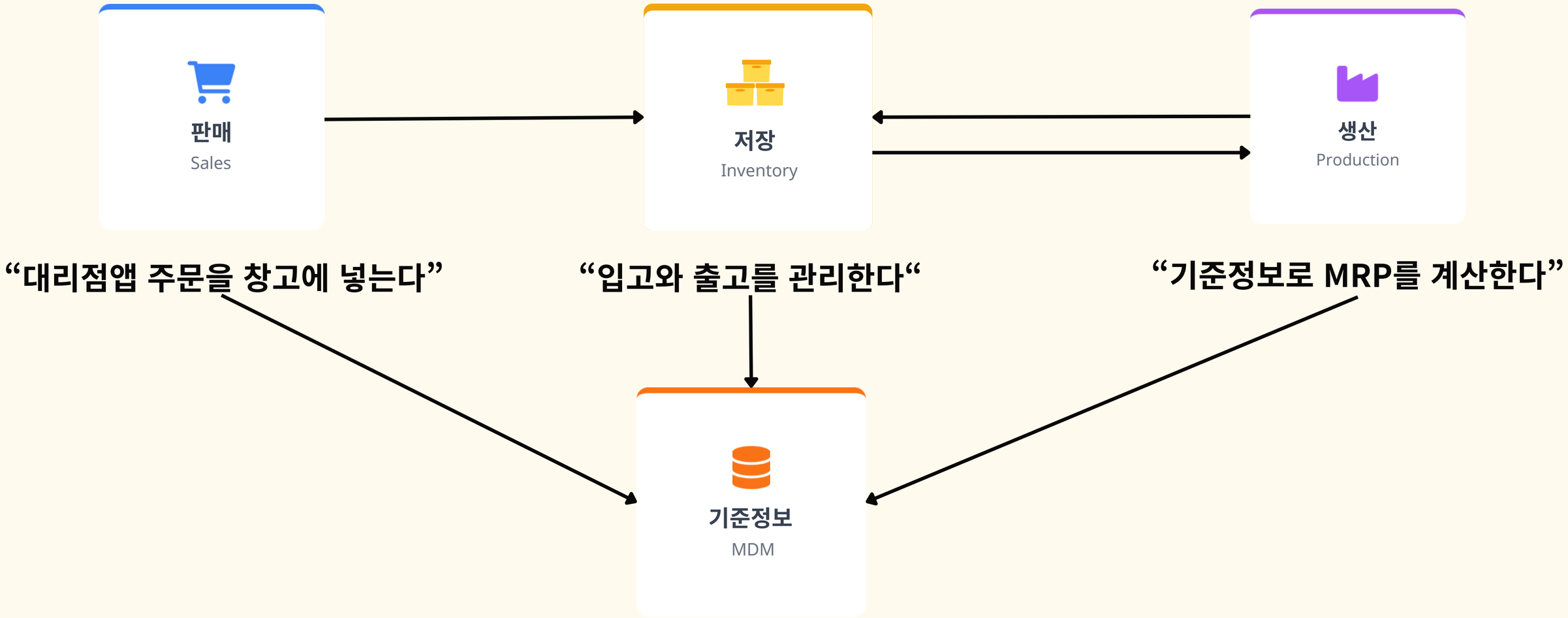
03 기획 및 설계 - WMS 워크플로우

03 기획 및 설계 - 생산 워크플로우



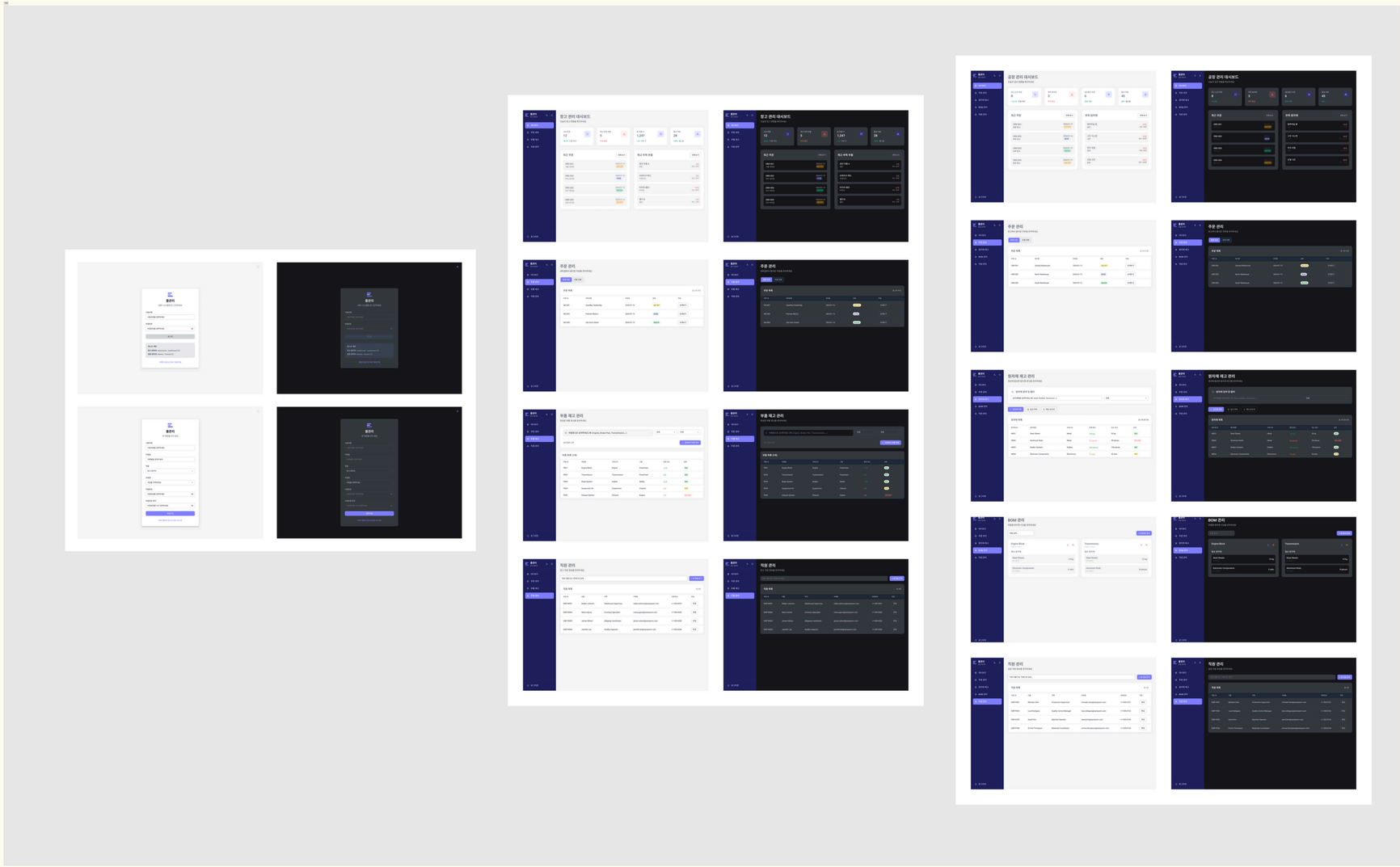
03 기획 및 설계 - 에자일 / MVP 기획 프로세스

핵심 도메인 식별

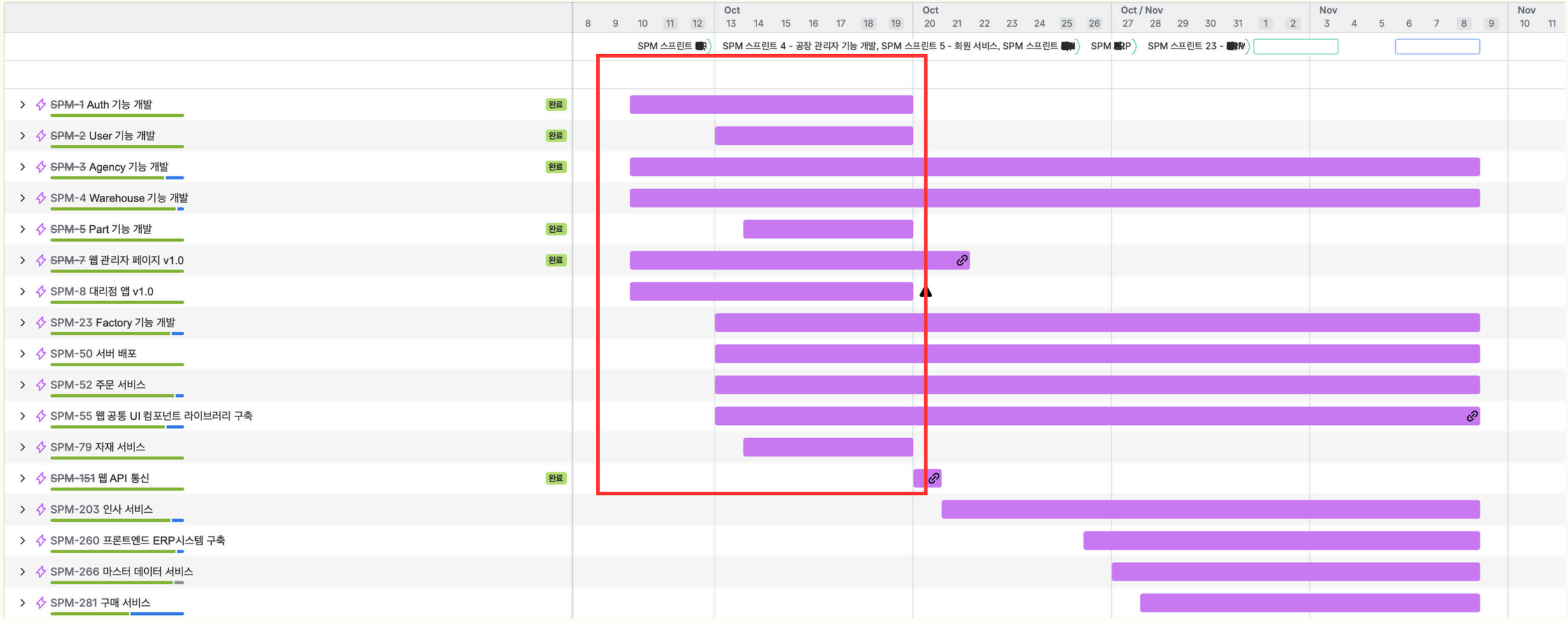


03 기획 및 설계 - 에자일 / MVP 기획 프로세스

디자인 초안

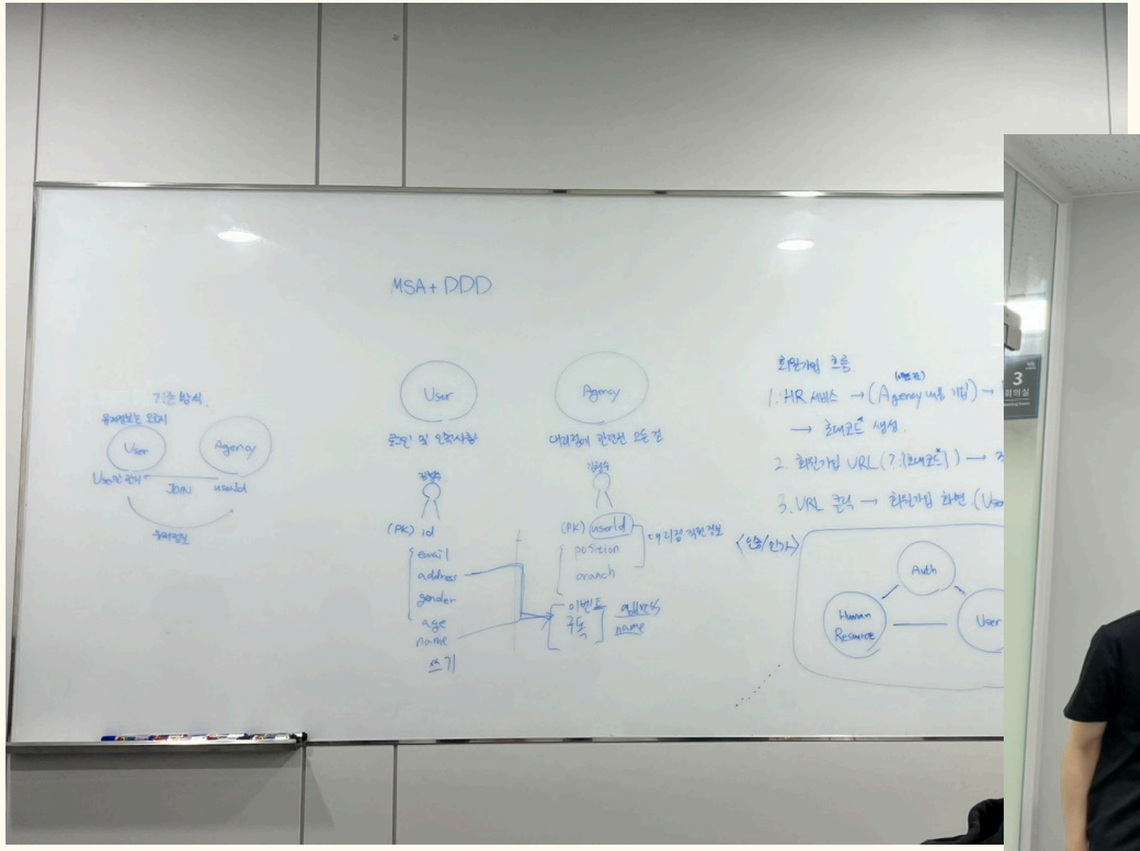


03 기획 및 설계 - 프로젝트 일정 및 개발 계획

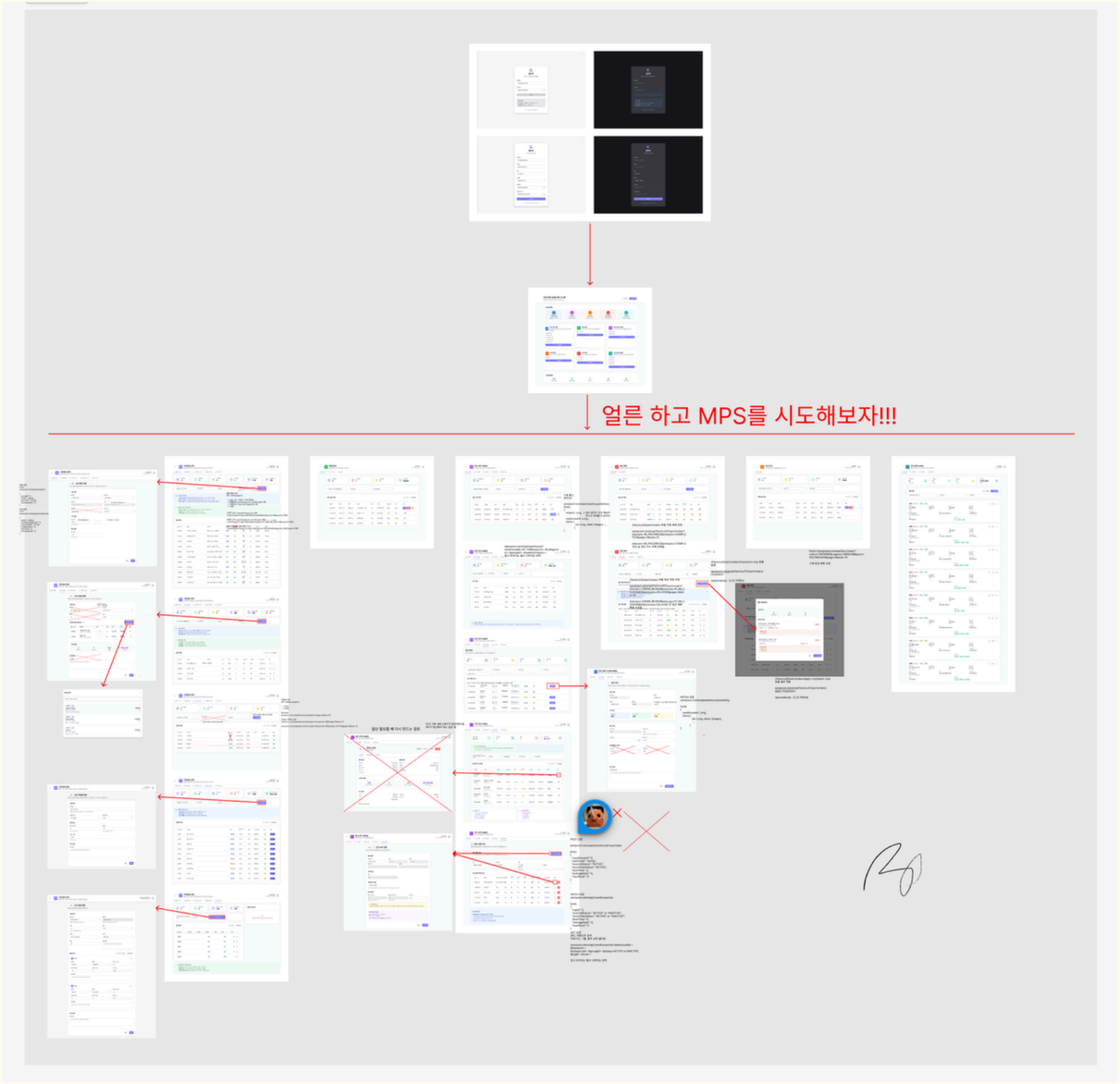


03 기획 및 설계 - 프로젝트 일정 및 개발 계획

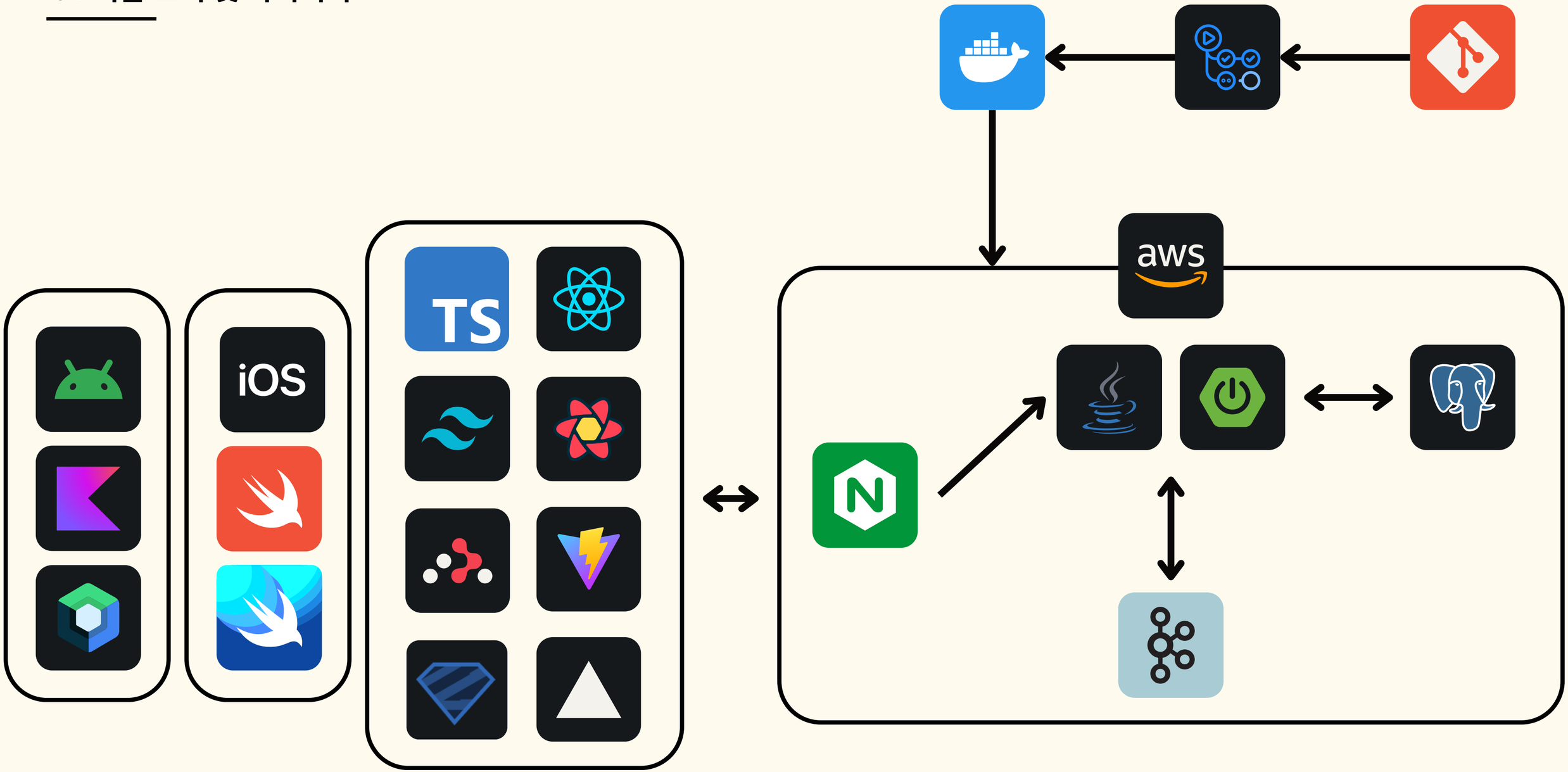
스프린트 중 세미나 (주 1~2회)



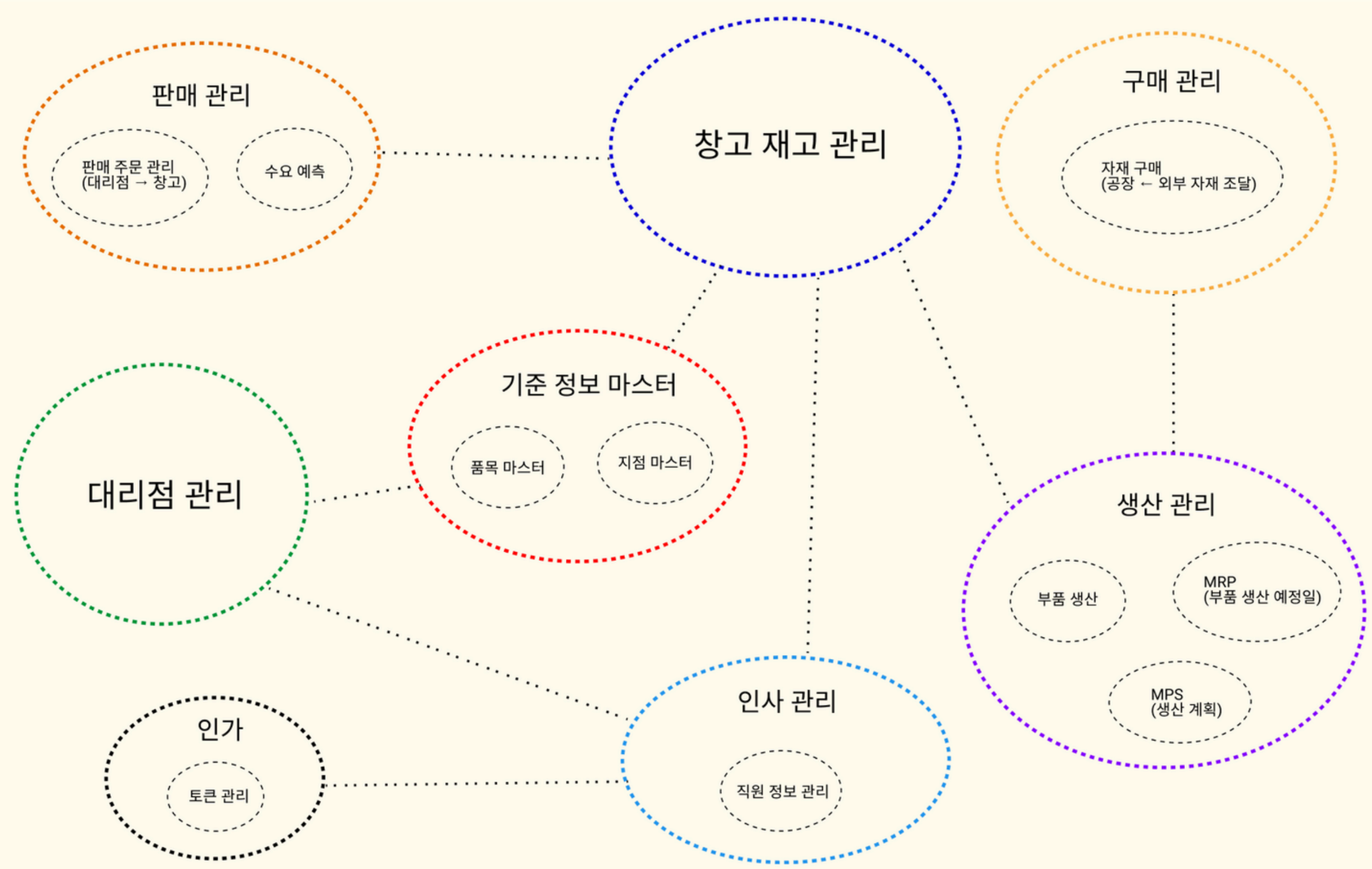
03 기획 및 설계 - UI/UX 디자인 시안



04 기술 스택 및 아키텍처



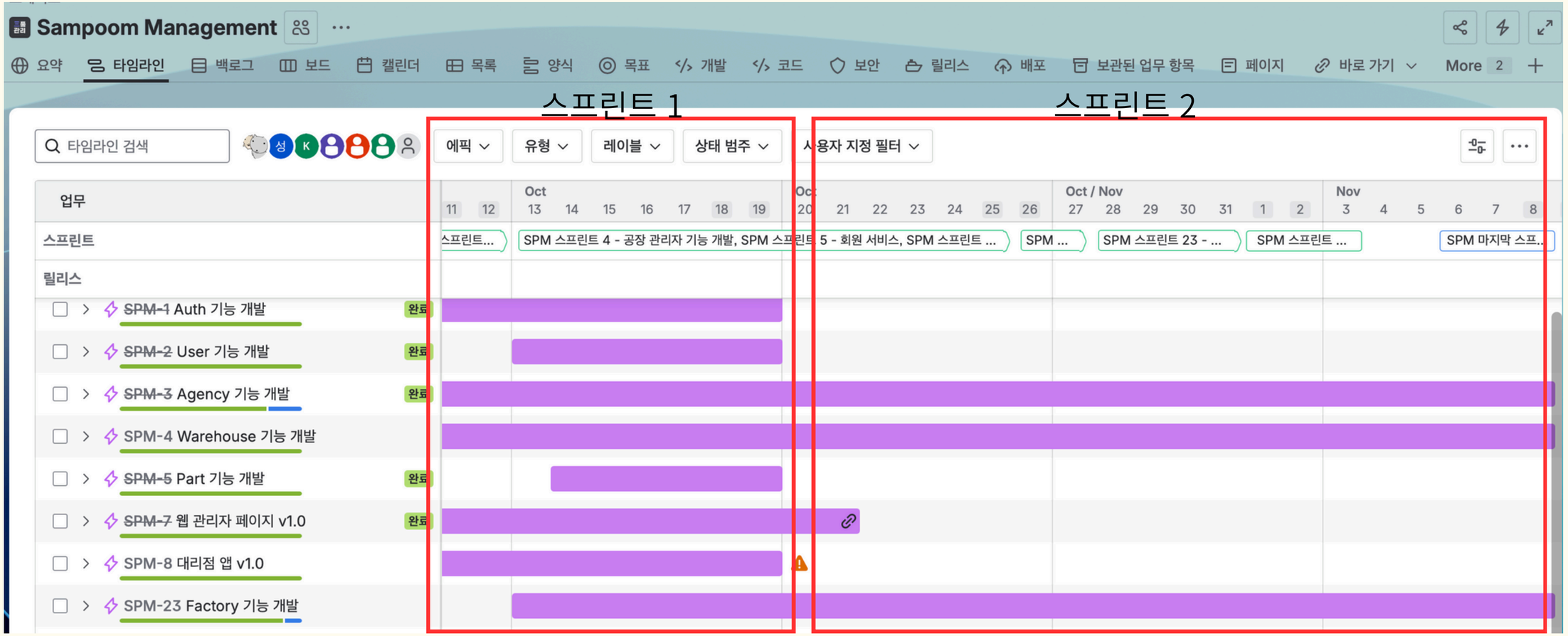
04 기술 스택 및 아키텍처



시연

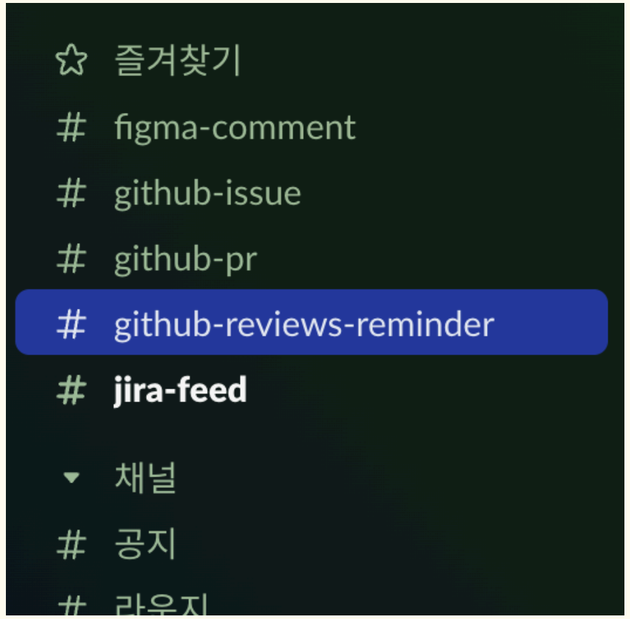
06 성과 및 차별점 - 협업

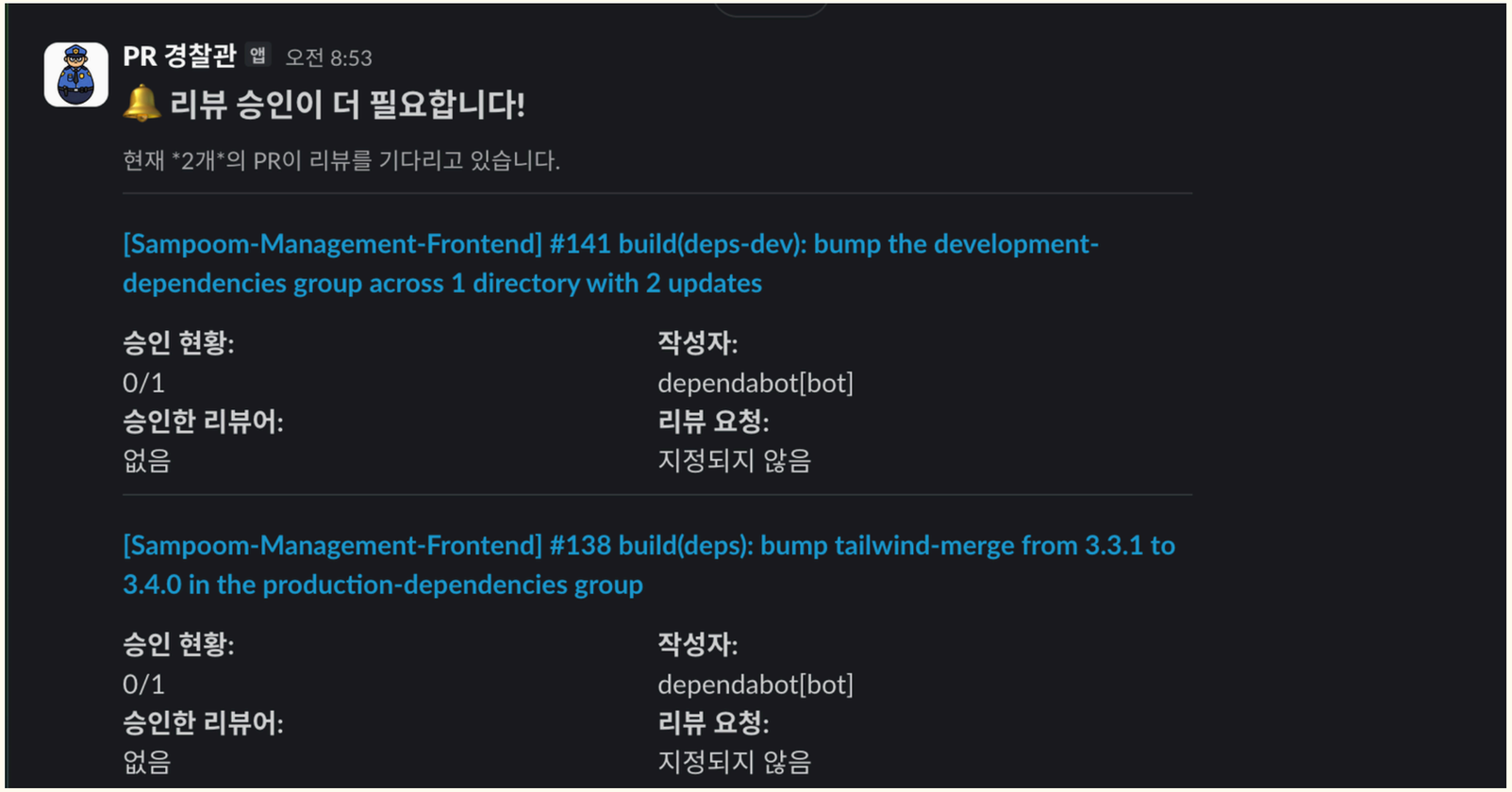
Jira



06 성과 및 차별점 - 협업

Slack





Kafka Outbox 패턴

배경

- 처음에는 DB 저장과 Kafka 전송을 한 트랜잭션에서 함께 처리하려 함
- DB 트랜잭션과 Kafka 전송은 별개로 동작

문제

- DB는 커밋됐지만 Kafka 전송이 실패할 수 있음
- Kafka는 전송됐지만 DB는 롤백될 수 있음

해결책

- DB 트랜잭션 안에서는 Kafka 대신 Outbox 테이블에 이벤트 기록
- 별도 프로세스가 Outbox 데이터를 Kafka로 안전하게 발행

공장장의 인생이란

Outbox 엔티티

```java
public class outbox {

    @Id
    @Column(name = "outbox_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;  // Outbox 식별자 (PK)

    @Column(nullable = false)
    private String eventType;  // 이벤트 종류 (예: FactoryCreated)

    @Column(nullable = false)
    private Long aggregateId;  // 관련 엔티티 ID (예: factory_id)

    @Column(name = "event_id", nullable = false, unique = true, c
    private UUID eventId;  // 이벤트 고유 ID (멱등성 보장용)
```

```java
    @JdbcTypeCode(SqlTypes.JSON)
    @Column(nullable = false, columnDefinition = "jsonb")
    private JsonNode payload;  // Kafka로 보낼 JSON 데이터

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 20)
    private OutboxStatus status;  // READY / PUBLISHED / FA

    @Column(nullable = false)
    private LocalDateTime occurredAt;  // 이벤트 발생 시각

    @Builder.Default
    @Column(nullable = false)
    private Integer retryCount = 0;

    @Column(columnDefinition = "text")
    private String lastError;

    private LocalDateTime publishedAt;

    private LocalDateTime lastTriedAt;

    private LocalDateTime nextRetryAt;
```

공장장의 인생이란

DB와 Kafka 이벤트 간 데이터 불일치

```
@Transactional
public Order createOrderBadWay(String productName, Integer quantity, String customerName) {
    try {
        Order order = new Order(productName, quantity, customerName);
        Order savedOrder = orderRepository.save(order);

        sendOrderCreatedEvent(savedOrder);

        return savedOrder;
    } catch (Exception e) {
        throw new RuntimeException("주문 생성에 실패했습니다", e);
    }
}
```

주문 생성 및 저장

즉시 카프카로 메시지 발송
발송 이후 예외가 발생하면

- DB 롤백은 되지만 이미 카프카 메시지는 발송됨
- 데이터 불일치 발생

07 개인 발표 - 김태민

공장장의 인생이란

```
@Transactional
public Order createOrder(String productName, Integer quantity, String customerName) {
    Order order = orderRepository.save(new Order(productName, quantity, customerName));


    OrderCreatedEvent event = new OrderCreatedEvent(order.getId(), productName, quantity, customerName);
    String payload = toJson(event);

    // 👉 트랜잭션 안에서는 Outbox에만 기록
    outboxRepository.save(Outbox.builder()
            .aggregateType("ORDER")
            .aggregateId(order.getId())
            .eventType("OrderCreated")
            .payload(payload)              // JSON 문자열
            .status(OutboxStatus.READY)   // 기본 READY
            .occurredAt(OffsetDateTime.now())
            .build());

    return order;
}
```

DB에 주문 데이터 저장 + Outbox 테이블에 이벤트 기록

- 같은 트랜잭션이기 때문에 데이터와 이벤트의 일관성이 보장

공장장의 인생이란

```java
@Component
@RequiredArgsConstructor
public class OrderOutboxPublisher {

    private final OutboxRepository outboxRepository;
    private final KafkaTemplate<String, String> kafkaTemplate;

    @Scheduled(fixedDelay = 500)
    public void publish() {
        List<Outbox> batch = outboxRepository.pickReadyBatch(100);
        for (Outbox o : batch) {
            try {
                markInProgress(o.getId());
                kafkaTemplate.send("order-events", String.valueOf(o.getAggregateId()), o.getPayload())
                             .get(5, TimeUnit.SECONDS);
                markPublished(o.getId());
            } catch (Exception ex) {
                markFailedOrDead(o.getId(), ex);
            }
        }
    }
}
```

Outbox Publisher

- 일정 주기마다 Outbox 테이블에서 아직 발행되지 않은 이벤트를 읽어 Kafka에 메시지를 전송하고, 전송 결과에 따라 상태를 관리하는 퍼블리셔 클래스.

07 개인 발표 - 성현주

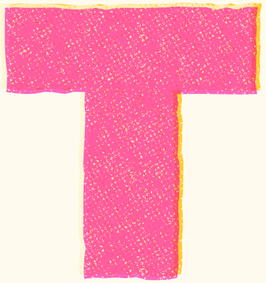
창고 관리자의 인생이란

```
@Transactional  1개 사용 위치  Admin
public void deliveryProcess(DeliveryReqDto deliveryReqDto) {
    Map<Long, Inventory> inventoryMap = this.getInventoryMap(
            deliveryReqDto.getWarehouseId(),
            deliveryReqDto.getItems()
    );
    PartUpdateReqDto partUpdateReqDto = new PartUpdateReqDto(
            deliveryReqDto.getWarehouseId(),
            deliveryReqDto.getItems());

    this.validateOutBound(partUpdateReqDto);
    this.updateParts(partUpdateReqDto, inventoryMap);
    this.saveOutHistory(deliveryReqDto.getItems(), inventoryMap);
    this.checkRop(deliveryReqDto);
    orderService.setOrderStatusEvent(deliveryReqDto.getOrderId(), OrderStatus.ARRIVED);
}
```

```
@Transactional  1개 사용 위치  Admin
protected void saveOutHistory(List<PartDeltaDto> items, Map<Long, Inventory> inventoryMap) {
```

Rabbit: 내부 호출 시 트랜잭션이 제대로 작동하지 않을 것이다.

창고 관리자의 인생이란



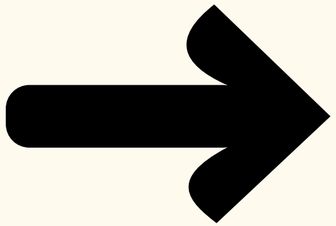
내부 호출 함수의 @Transactional 문제

같은 함수를 내부 호출 / 외부 호출을 따로 둬야 한다?

07 개인 발표 - 성현주

창고 관리자의 인생이란

```java
@Transactional  1개 사용 위치
public void updateStock() {
    System.out.println("=== 재고 업데이트 시작 ===");

    Stock stock = new Stock();
    stock.setItemName("engine");
    stock.setQuantity(10);
    stockRepository.save(stock);

    // 내부 호출 (프록시 안 거침)
    saveHistory();

    System.out.println("=== 재고 업데이트 완료 ===");
}

@Transactional  1개 사용 위치
public void saveHistory() {
    System.out.println("=== 재고 이력 저장 ===");

    StockHistory history = new StockHistory();
    history.setAction("update");
    stockHistoryRepository.save(history);

    System.out.println("❌ 예외 발생");
    throw new RuntimeException("아무튼 예외입니다");
}
```



```
=== 재고 업데이트 시작 ===
Hibernate: insert into stock (item_name,quantity,id) values (?,?,default)
=== 재고 이력 저장 ===
Hibernate: insert into stock_history (action,id) values (?,default)
❌ 예외 발생
Exception in thread "main" java.lang.RuntimeException Create breakpoint : 아무튼 예외입니다
```

```
select * from stock;
```

QUANTITY	ID	ITEM_NAME

(no rows, 6 ms)

```
select * from stock_history;
```

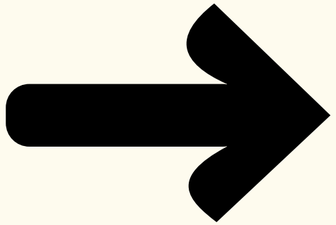
ID	ACTION

(no rows, 8 ms)

07 개인 발표 - 성현주

창고 관리자의 인생이란

```java
public void updateStock() {  1개 사용 위치
    System.out.println("=== 재고 업데이트 시작 ===");

    Stock stock = new Stock();
    stock.setItemName("engine");
    stock.setQuantity(10);
    stockRepository.save(stock);

    // 내부 호출 (프록시 안 거침)
    saveHistory();

    System.out.println("=== 재고 업데이트 완료 ===");
}

@Transactional  1개 사용 위치
public void saveHistory() {
    System.out.println("=== 재고 이력 저장 ===");

    StockHistory history = new StockHistory();
    history.setAction("update");
    stockHistoryRepository.save(history);

    System.out.println("❌ 예외 발생");
    throw new RuntimeException("아무튼 예외입니다");
}
```



```
=== 재고 업데이트 시작 ===
Hibernate: insert into stock (item_name,quantity,id) values (?,?,default)
=== 재고 이력 저장 ===
Hibernate: insert into stock_history (action,id) values (?,default)
❌ 예외 발생
Exception in thread "main" java.lang.RuntimeException Create breakpoint : 아무튼 예외입니다
```

select * from stock;

QUANTITY	ID	ITEM_NAME
10	1	engine

(1 row, 4 ms)

select * from stock_history;

ID	ACTION
1	update

(1 row, 1 ms)

A

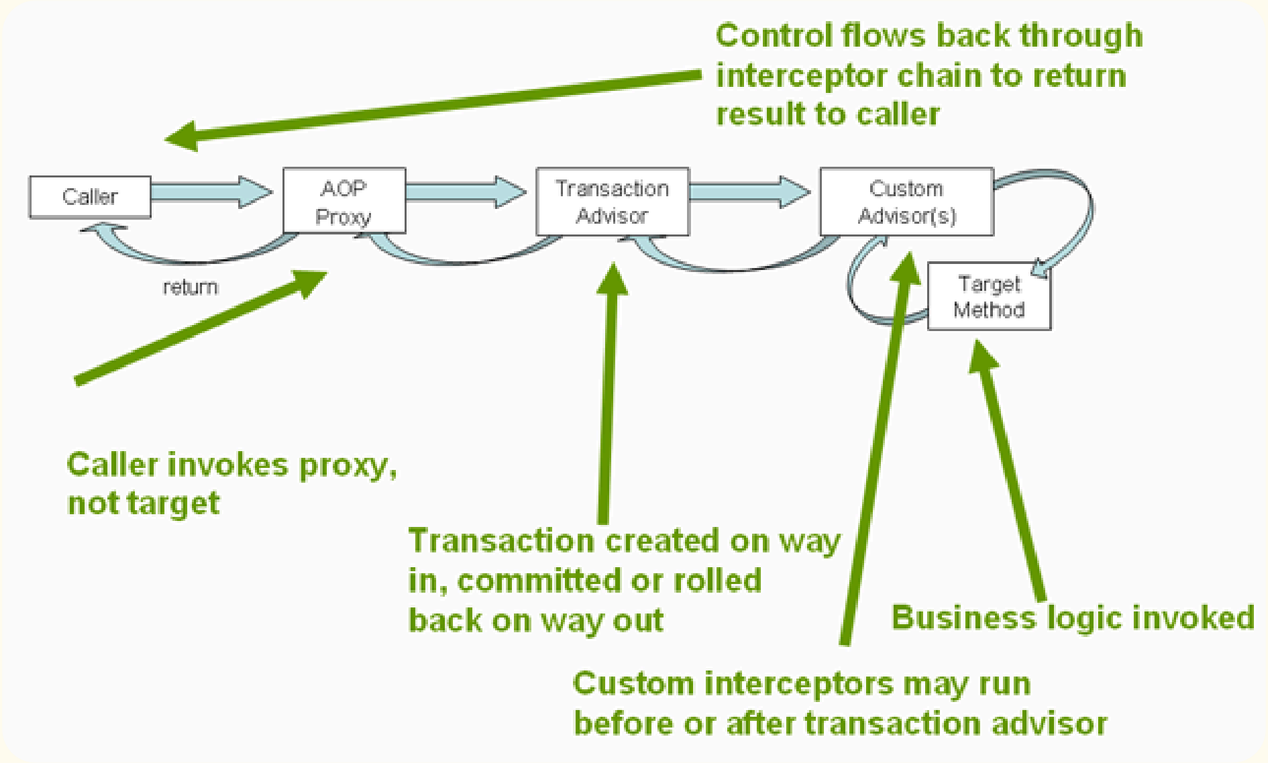
AOP - Aspect Orient Programming

Aspect: 여러 클래스에 적용되는 관심사의 모듈화

Proxy: Aspect를 구현하기 위한 객체

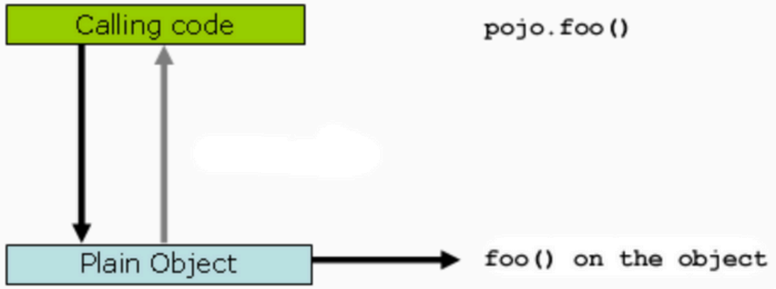
07 개인 발표 - 성현주

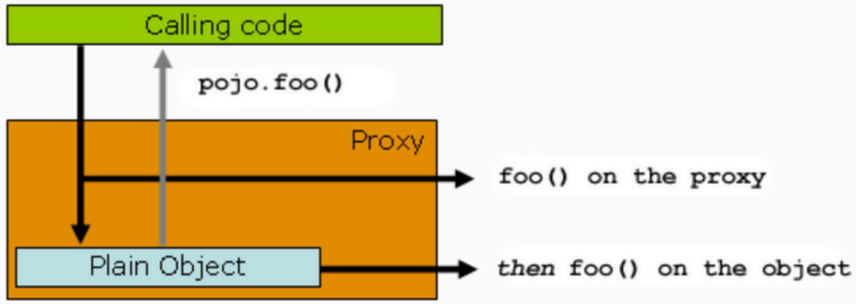
창고 관리자의 인생이란



07 개인 발표 - 성현주

창고 관리자의 인생이란





NOTE

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation (in effect, a method within the target object calling another method of the target object) does not lead to an actual transaction at runtime even if the invoked method is marked with `@Transactional`. Also, the proxy must be fully initialized to provide the expected behavior, so you should not rely on this feature in your initialization code—for example, in a `@PostConstruct` method.

07 개인 발표 - 성현주

창고 관리자의 인생이란

내부 호출 함수의 @Transactional은 무시될 뿐 에러가 아니다

같은 함수를 내부 호출 / 외부 호출을 따로 둬야 한다? 아니다

p.s AI의 말은 한번에 믿지 말자

지점장의 인생이란

QueryDSL

배경

- 엔티티 간 연관관계 없음 → categoryId, groupId, partId만 보유
- “카테고리 → 그룹 → 부품” 구조 + 재고/출고 수량 표시 계층형 응답

문제

- JPA 기본 메서드로는 복잡한 다중 조인 처리 불가
- 문자열 JPQL은 오류 발생 가능성이 높음

해결책

- SQL 조인을 명시적 코드 형태로 표현 가능
- IDE 자동완성 + 컴파일 타임 검증으로 안정성 확보

07 개인 발표 - 양지선

지점장의 인생이란

```
    @Query("""  0개의 사용위치  신규 *
SELECT p FROM Part p
JOIN PartGroup g ON p.groupId = g.id
JOIN Category c ON g.categoryId = c.id
WHERE (LOWER(p.name) LIKE %:kw%)
""")
    List<Part> findParts(@Param("keyword") String keyword);
```

- 문자열 기반이라 컬럼 변경 시 오류 위험
- 조인 구조 복잡 → 유지보수 어려움
- 동적 조건(재고, 상태 등) 추가 어려움
- 엔티티 간 연관관계 없으면 JOIN 불가능

```
public List<AgencyCartResponseDTO> findCartItemsByUserId(String userId)
    QAgencyCartItem cart = QAgencyCartItem.agencyCartItem;
    QPart part = QPart.part;
    QPartGroup group = QPartGroup.partGroup;
    QCategory category = QCategory.category;
```

엔티티 기반으로 생성되는 QPart, QCategory 등의 Q클래스가 SQL 테이블/컬럼 역할을 대신한다

- 타입 안전성: 필드명 변경 시 컴파일 단계에서 오류를 감지해 안정적인 개발 환경 구축
- IDE 지원: 메서드 체인 방식으로 가독성 및 생산성 향상

07 개인 발표 - 양지선

지점장의 인생이란

```
queryFactory
 .select(new QAgencyCartResponseDTO(
         cart.id,
         part.id,
         part.name,
         part.code,
         cart.quantity,
         group.id,
         group.name,
         category.id,
         category.name,
         part.standardCost
 ))
 .from(cart)
 .join(part).on(cart.partId.eq(part.id))
 .leftJoin(group).on(part.groupId.eq(group.id))
 .leftJoin(category).on(group.categoryId.eq(category.id))
 .where(cart.userId.eq(userId))
 .fetch();
```

DTO Projection을 통한 성능 최적화

- select(new QDTO(...)) 구문을 사용하여 JPA 엔티티 대신 필요한 컬럼만 DTO로 조회하여 불필요한 연관관계 제거

연관관계 없는 테이블 간의 다중 JOIN 구현

- join().on()을 사용하여 엔티티 매핑과 관계없이 SQL처럼 명시적으로 4개 테이블을 JOIN

```
.from(part)
.join(group).on(part.groupId.eq(group.id))
.join(category).on(group.categoryId.eq(category.id))
.leftJoin(stock).on(stock.partId.eq(part.id)
        .and(stock.agency.id.eq(agencyId)))
.where(
        keyword != null && !keyword.isBlank()
                ? part.name.containsIgnoreCase(keyword)
                .or(part.code.containsIgnoreCase(keyword))
                : null
)
.offset(pageable.getOffset())
.limit(pageable.getPageSize())
.fetch();
```

복합 조건 JOIN

- stockId.eq(...).and(...)와 같이 두 가지 이상의 조건으로 LEFT JOIN을 처리하여 재고 조회 문제를 해결

동적 WHERE 절

- keyword != null ? : null과 같이 자바 코드로 검색 조건의 유무를 판단하여 동적 쿼리 처리 가능

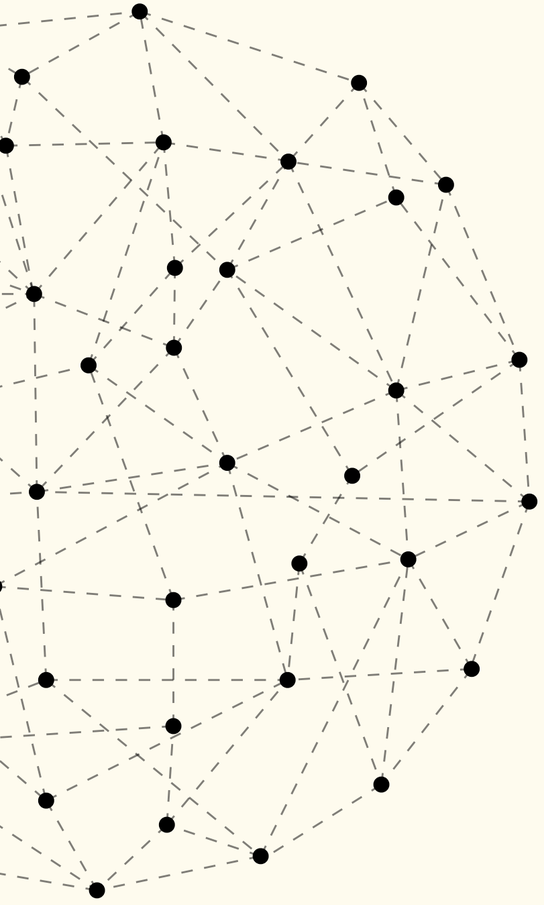
지점장의 인생이란

```java
public <T extends PartFlatDTO> List<CategoryResponseDTO> toNestedStructure(List<T> items)

    // 카테고리 단위로 그룹핑
    Map<Long, List<T>> byCategory = items.stream()
            .collect(Collectors.groupingBy(T::getCategoryId));

    List<CategoryResponseDTO> categories = new ArrayList<>();

    for (Map.Entry<Long, List<T>> categoryEntry : byCategory.entrySet()) {
        List<T> categoryItems = categoryEntry.getValue();

        // 각 카테고리 내에서 그룹 단위로 그룹핑
        Map<Long, List<T>> byGroup = categoryItems.stream()
                .collect(Collectors.groupingBy(T::getGroupId));
```

Java Stream을 활용한 계층 구조 변환

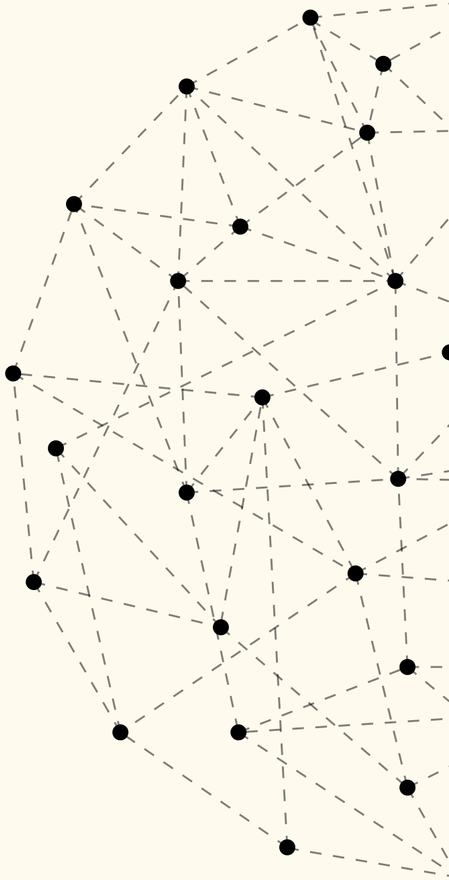
- QueryDSL로 얻은 Flat DTO 결과를 카테고리 → 그룹 → 부품 형태의 Nested Structure로 최종 변환
- Java Stream의 Collectors.groupingBy()를 사용하여 메모리 상에서 효율적으로 데이터를 가공

인사 팀장의 인생이란



“ MSA, 다시 선을 긋다. ”

MSA 구조 속 책임과 경계를 재정의한 리팩터링 경험



인사 팀장의 인생이란

초기 시스템은 MSA 구조로 구성된 ERP 플랫폼

기능 별로 독립된 서비스로 분리

기능 별로 독립된 서비스로 분리

관리하는 조직 별 직원 정보를 별도의 DB로 관리

인증·인가를 담당하는 Auth, 인사 관리를 담당하는 User

회원가입 · 로그인 시 어느 조직 직원으로 인증할지 판단 필요

인사 팀장의 인생이란

당시 상황

Gateway 통합 인증/인가가 아닌 **각 서비스별 API 인증/인가**
API 호출 시마다 인증자의 소속 조회 필요 → 비용 발생

```
@PatchMapping("/role/{userId}")  0개의 사용위치  LeeJongJin
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
@Operation(summary = "권한 변경", description = "특정 유저의 접근 권한을 변경합
public ResponseEntity<ApiResponse<RoleResponse>> updateRole(
        Authentication authentication,
        @PathVariable Long userId,
        @RequestBody RoleRequest roleRequest
```

인사 팀장의 인생이란

설계 목표

CRUD 가능 권한과 별도로,
토큰 내 정보에 **소속 조직 별로 접근 권한**을 추가해서 **인가**하는 구조적 설계로 개선

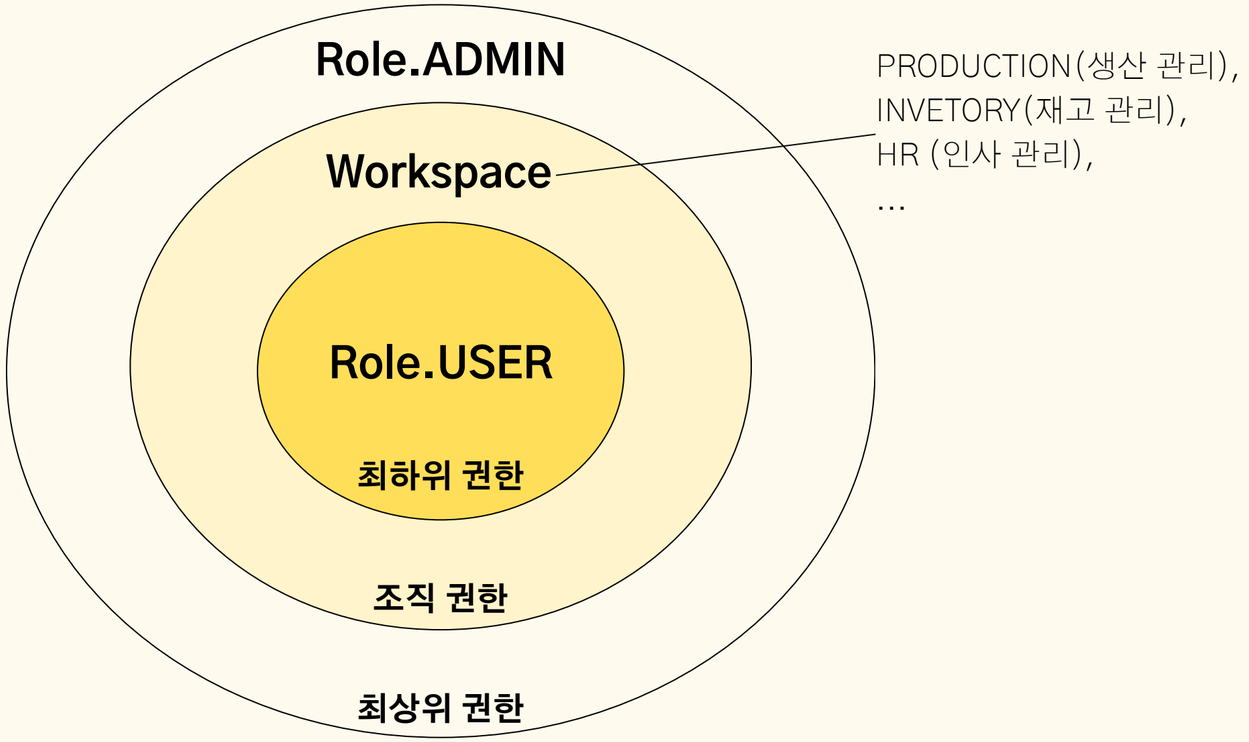
```
@PreAuthorize("hasAuthorize('ROLE_PURCHASE')")  0개의 사용위치  Kim Taemin
@Operation(summary = "자재 주문 생성", description = "공장에 필요한 자재 주문을 생성합니다.
@PostMapping()
public ResponseEntity<ApiResponse<PurchaseOrderResponseDto>> createMaterialOrde
        @RequestBody PurchaseOrderRequestDto requestDto) {
    return ApiResponse.success(SuccessStatus.CREATED,
            purchaseService.createMaterialOrder(requestDto));
}
```

07 개인 발표 - 이종진

인사 팀장의 인생이란

설계 목표

Role: CRUD 권한 (USER, ADMIN)
Workspace: 소속된 조직 접근 권한



→ 인가 검증 시 **CRUD 권한**과 **조직 접근 권한**을 **동시에 비교**

```
Role role;
Workspace workspace;
try {
    role = Role.valueOf(roleStr);
    workspace = Workspace.valueOf(workspaceStr);
} catch (IllegalArgumentException ex) {
    throw new CustomAuthenticationException(ErrorStatus.
}

// 권한 매핑 (Enum Role → Security 권한명)
String roleAuthority = "ROLE_" + role.name();
String workspaceAuthority = "ROLE_" + workspace.name();

// GrantedAuthority 리스트 생성
List<GrantedAuthority> authorities = new ArrayList<>();
authorities.add(new SimpleGrantedAuthority(roleAuthority
authorities.add(new SimpleGrantedAuthority(workspaceAuth

UsernamePasswordAuthenticationToken authentication = new
```

인증 객체에 **Role**과 **Workspace**를 함께 등록

인사 팀장의 인생이란

MSA의 핵심 특징, **단일 진실 공급원**
→ Auth는 인증과 인가에만 집중, User는 사용자의 소속을 명확히 관리

문제 발생

초기 설계 시 소속 데이터는 **인가**를 맡은 **Auth**보다는
직원 정보 책임의 **User**쪽 책임으로 설계
→ **책임의 경계**를 다르게 설정

인사 팀장의 인생이란

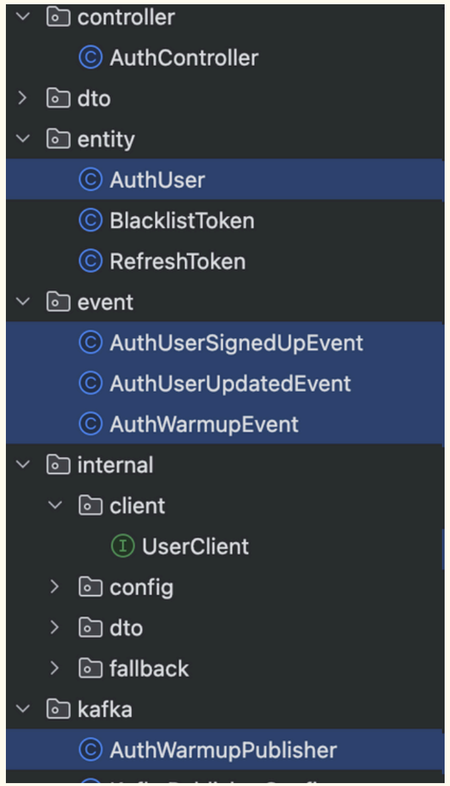
해결 과정

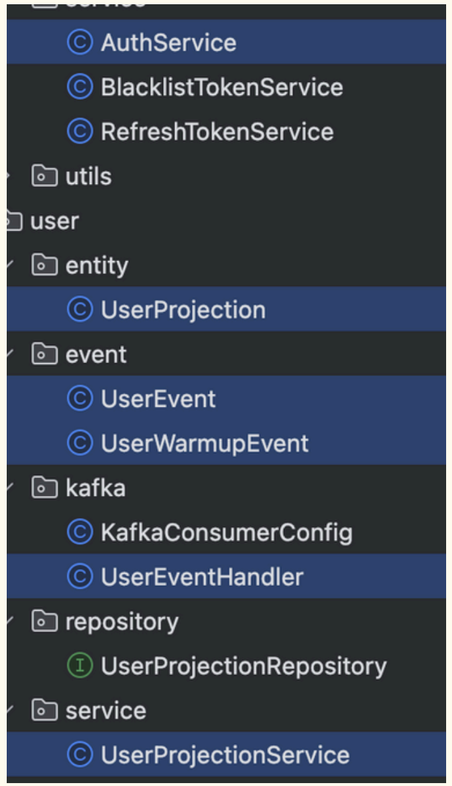
인증/인가 정보를 다루는 Auth가 소속 접근 권한 직접 소유
→ **Auth로의 이관** 작업 필요

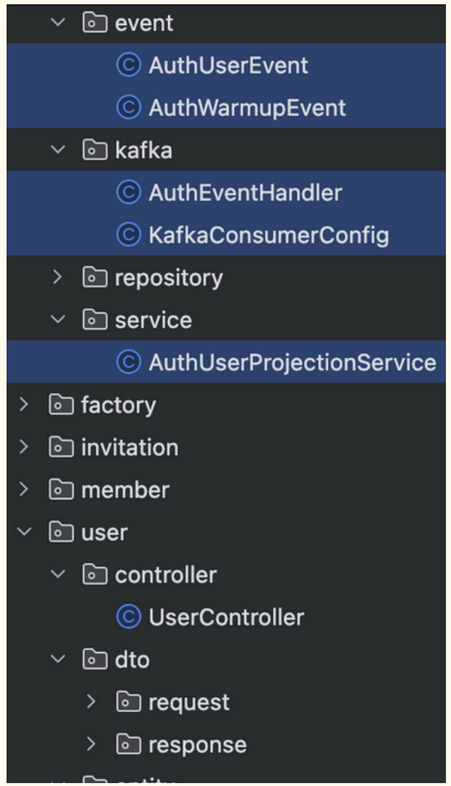
모듈 내 응집도가 높은 MSA의 특성상 DB의 컬럼 하나를 이전하는데
내부 복잡도 및 **책임 분리 검증**에 상당한 노력과 시간 소요
→ **유지보수 비용 증가**

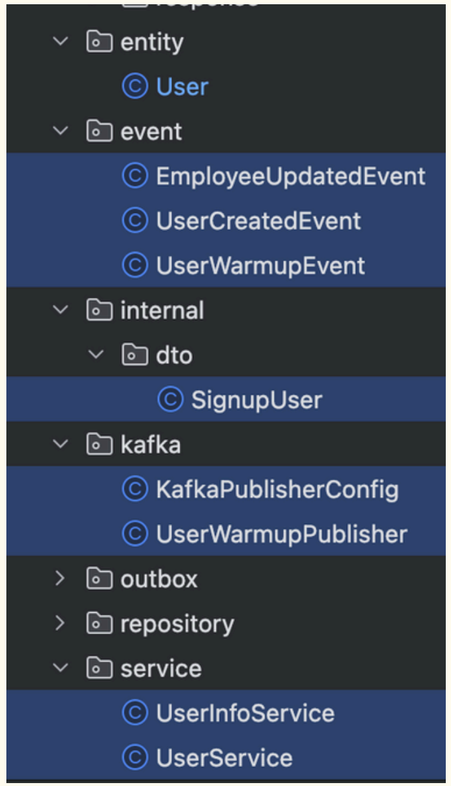
07 개인 발표 - 이종진

인사 팀장의 인생이란







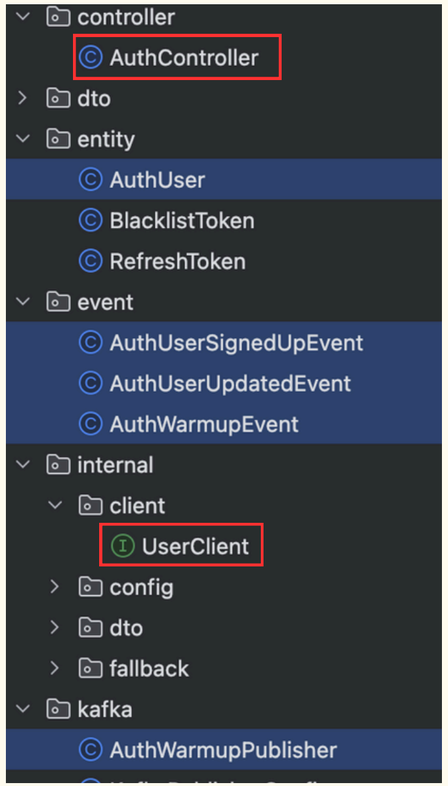


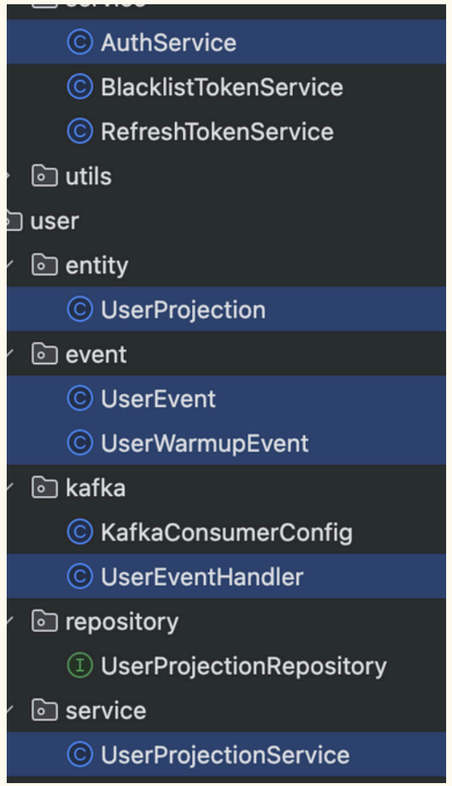
Auth 서버 디렉토리

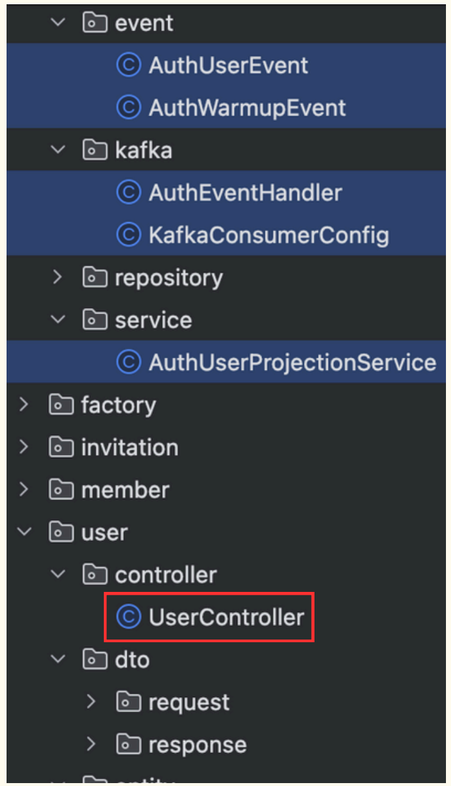
User 서버 디렉토리

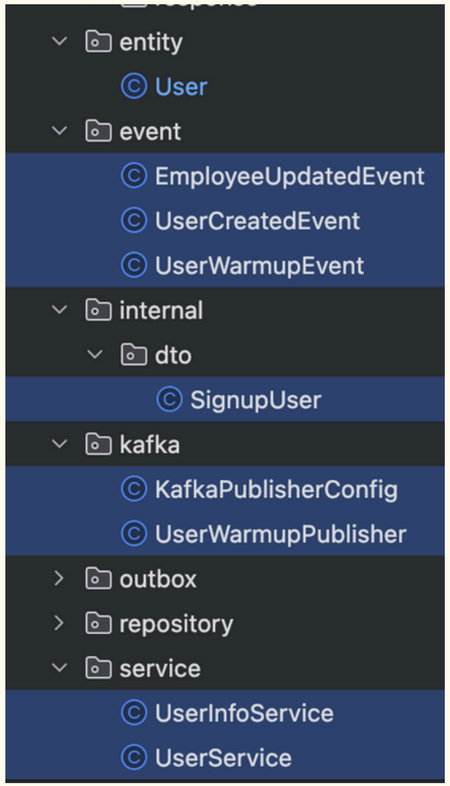
07 개인 발표 - 이종진

인사 팀장의 인생이란









Auth 서버 디렉토리

User 서버 디렉토리

인사 팀장의 인생이란

결과 분석

상당히 변경된 **내부 구조**와 **비즈니스 로직**, 클라이언트 및 서비스 간 **외부 API는 그대로**
→ 서비스 **내부 응집도**가 **높고**, **외부 결합도**가 **낮다**는 MSA의 핵심 특징

MSA에서 중요한 건 분리 그 자체가 아니라
어떻게 **명확하게 분리**할지에 대한 **책임의 일관성**이 중요

클린 아키텍처

동시 개발

배경

- 서로 다른 언어(Kotlin / Swift)로 작성되어 코드 재사용 불가
- 하지만 비즈니스 로직은 동일해야 함

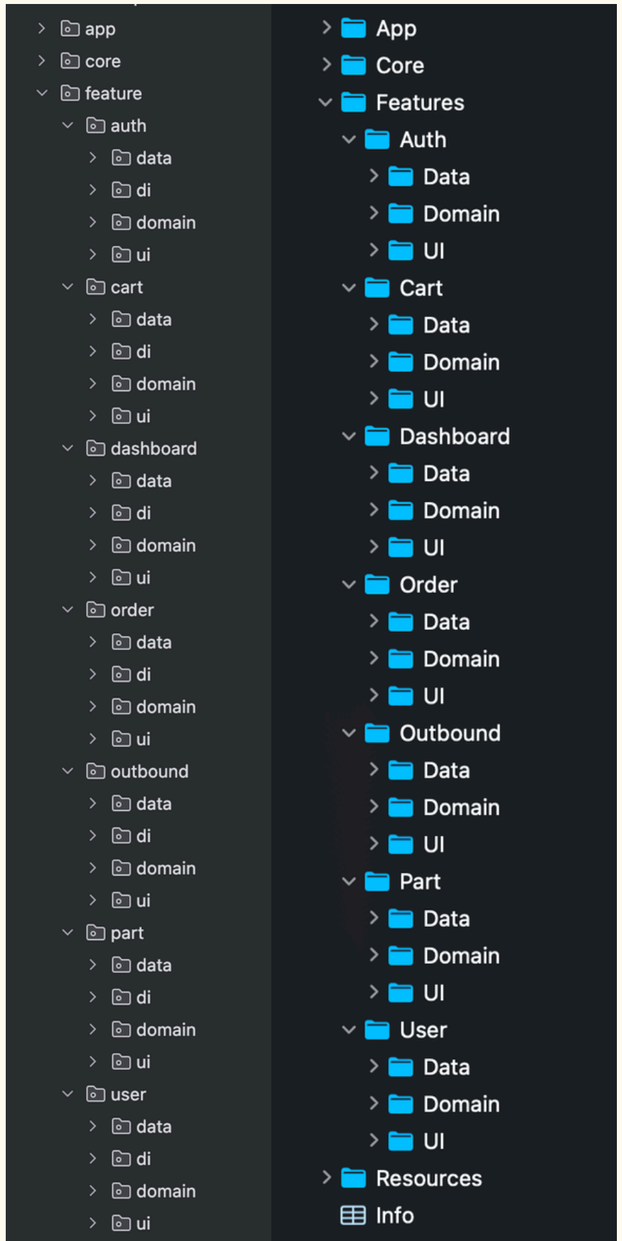
문제

- 기능이 늘어나면서 플랫폼별로 로직 불일치 발생 위험
- UI 프레임워크나 라이브러리 교체 시 비즈니스 코드까지 수정 필요

해결책

- 두 플랫폼이 같은 도메인 구조와 규칙을 공유하도록 설계
- 클린 아키텍처 도입

07 개인 발표 - 채상윤



클린 아키텍처

클린 아키텍처?

비즈니스 로직(도메인)을 중심에 두고, 외부 요소(UI, DB, 네트워크)에 대한 의존성을 안쪽(도메인)으로만 흐르게 하는 설계 구조

계층	알 수 있는 계층	알 수 없는 계층
Domain	없음	UI, Data
Data	Domain	UI
UI	Domain	Data

Domain Layer

핵심 비즈니스 로직
플랫폼이나 프레임워크에 의존하지 않음

UI(Presentation) Layer

사용자 인터페이스
사용자의 입력과 화면 상태를 관리하는 계층

Data Layer

실제 데이터 처리
외부 세계(DB, API, 네트워크 등)와 연결되는 계층

07 개인 발표 - 채상윤

클린 아키텍처

Domain Layer

핵심 비즈니스 로직
플랫폼이나 프레임워크에
의존하지 않음

Domain Model - 도메인에서 다루는 실제 엔티티

```
data class VendorList(
    val items: List<Vendor>,
    val totalCount: Int = items.size,
    val isEmpty: Boolean = items.isEmpty()
)
```

Repository Interface - 데이터 접근 방법의 약속을 정의

```
interface AuthRepository {
    suspend fun signUp(...): Result<User>
    suspend fun signIn(email: String, password: String): Result<User>
    suspend fun signOut(): Result<Unit>
    // ...
}
```

외부 구현을 몰라도 동작할 수 있는
앱의 중심 계층

UseCase - 하나의 기능 단위 로직을 담당

```
class LoginUseCase @Inject constructor(
    private val repository: AuthRepository
) {
    suspend operator fun invoke(email: String, password: String): Result<User> = repository.signIn(email, password)
}
```

07 개인 발표 - 채상윤

```
struct VendorList: Equatable {
    let items: [Vendor]
    var totalCount: Int { items.count }
    var isEmpty: Bool { items.isEmpty }
    static func empty() -> VendorList { VendorList(items: []) }
}
```

```
protocol AuthRepository {
    func signUp(...) async throws -> User
    func signIn(email: String, password: String) async throws -> User
    func signOut() async throws
    // ...
}
```

```
class LoginUseCase {
    private let repository: AuthRepository

    init(repository: AuthRepository) {
        self.repository = repository
    }

    func execute(email: String, password: String) async throws -> User {
        return try await repository.signIn(email: email, password: password)
    }
}
```

클린 아키텍처

Domain Layer

핵심 비즈니스 로직
플랫폼이나 프레임워크에
의존하지 않음

07 개인 발표 - 채상윤

클린 아키텍처

Data Layer

실제 데이터 처리
외부 세계(DB, API, 네트워크 등)와 연결되는 계층

API / DTO - 외부 데이터 형식을 다룸

```
interface AuthApi {
    @POST("auth/login")
    suspend fun login(@Body body: LoginRequestDto): ApiResponse<LoginResponseDto>
    // ...
}
```

RepositoryImpl - 도메인의 Repository 인터페이스 구현체로 서버,DB 연결

```
class AuthRepositoryImpl @Inject constructor(
    private val api: AuthApi,
    private val preferences: AuthPreferences
) : AuthRepository {
    override suspend fun signIn(email: String, password: String): Result<User> {
        return runCatching {
            val loginDto = api.login(LoginRequestDto(...))
            val loginUser = loginDto.data.toModel()
            preferences.saveUser(loginUser)
            loginUser
        }
    }
}
```

Mappers - 외부 데이터(DTO)와 도메인 모델 변환 계층 → 덕분에 계층간 의존성 분리

```
fun LoginResponseDto.toModel(): User = User(
    userId = userId,
    accessToken = accessToken,
    refreshToken = refreshToken,
    // ...
)
```

07 개인 발표 - 채상윤

클린 아키텍처

UiState - 화면의 상태를 데이터 형태로 관리

```
data class LoginUiState(
    val email: String = "",
    val password: String = "",
    val emailError: String? = null,
    val passwordError: String? = null,
    val loading: Boolean = false,
    val success: Boolean = false
) {
    val isValid: Boolean
        get() = email.isNotBlank() && password.isNotBlank() && ...
}
```

UiEvent - 사용자의 액션 표현

```
sealed interface LoginUiEvent {
    data class EmailChanged(val email: String) : LoginUiEvent
    data class PasswordChanged(val password: String) : LoginUiEvent
    data object Submit: LoginUiEvent
}
```

UI(Presentation) Layer

사용자 인터페이스
사용자의 입력과 화면 상태를 관리하는 계층

07 개인 발표 - 채상윤

ViewModel - UI의 상태를 관리, 도메인 유스케이스 호출해서 데이터 다룸

```
@HiltViewModel
class LoginViewModel @Inject constructor(
    private val loginUseCase: LoginUseCase,
    private val getProfileUseCase: GetProfileUseCase
) : ViewModel() {
    // UiState 관리
    private val _uiState = MutableStateFlow(LoginUiState())
    val uiState: StateFlow<LoginUiState> = _uiState

    // UiEvent 처리
    fun onEvent(e: LoginUiEvent) = when (e) {
        is LoginUiEvent.EmailChanged -> {
            _uiState.value = _uiState.value.copy(email = e.email)
            validateEmail()
        }
        is LoginUiEvent.PasswordChanged -> {
            _uiState.value = _uiState.value.copy(password = e.password)
            validatePassword()
        }
        is LoginUiEvent.Submit -> submit()
    }

    private fun submit() = viewModelScope.launch {
        if (!_uiState.value.isValid) return@launch

        _uiState.update { it.copy(loading = true) }
        loginUseCase(_uiState.value.email, _uiState.value.password)
            .onSuccess { ... }
            .onFailure { ... }
    }
}
```

클린 아키텍처

UI(Presentation) Layer

사용자 인터페이스
사용자의 입력과 화면 상태를 관리하는 계층

07 개인 발표 - 채상윤

클린 아키텍처

View - 오직 UI를 그리고 상태만 관찰

```
@Composable
fun LoginScreen(
    viewModel: LoginViewModel = hiltViewModel()
) {
    // UiState 구독
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column {
        // UiEvent 발생: EmailChanged
        CommonTextField(
            value = uiState.email,
            onValueChange = { viewModel.onEvent(LoginUiEvent.EmailChanged(it)) },
            isError = uiState.emailError != null,
            errorMessage = uiState.emailError
        )

        // UiEvent 발생: PasswordChanged
        CommonTextField(
            value = uiState.password,
            onValueChange = { viewModel.onEvent(LoginUiEvent.PasswordChanged(it)) },
            isPassword = true,
            isError = uiState.passwordError != null,
            errorMessage = uiState.passwordError
        )
        // ...
    }
}
```

UI(Presentation) Layer

사용자 인터페이스
사용자의 입력과 화면 상태를 관리하는 계층

클린 아키텍처

동시 개발

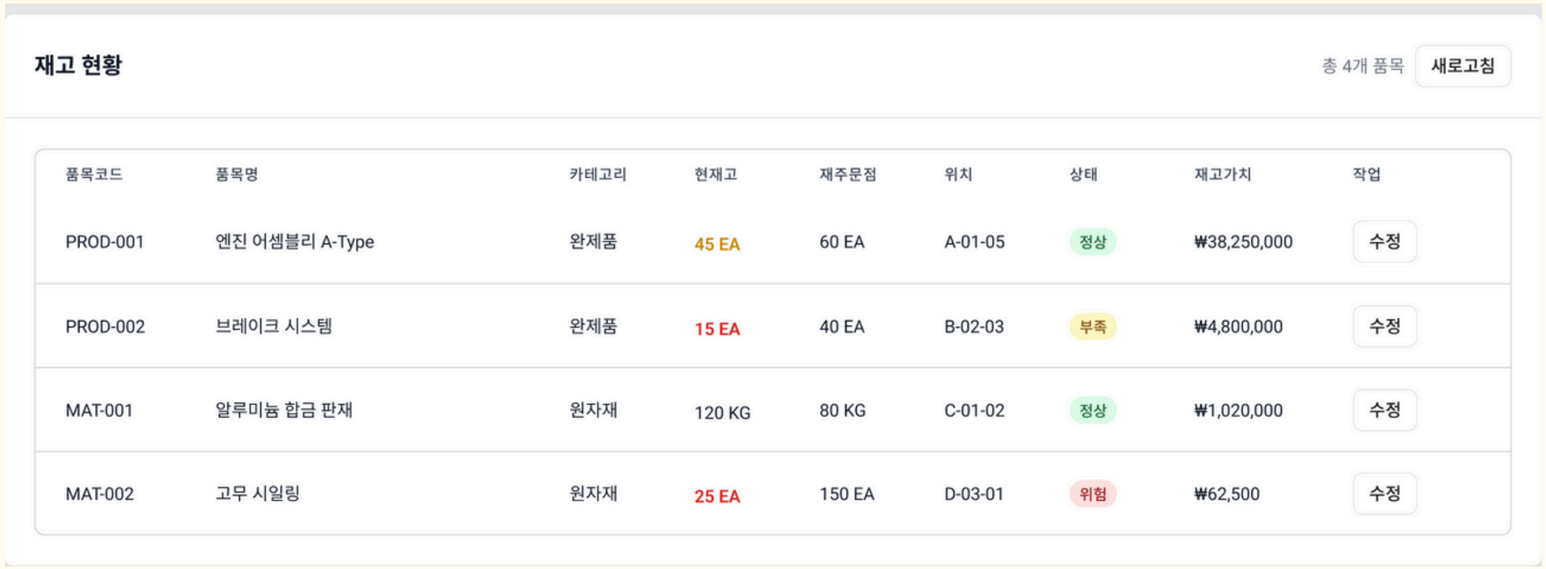
결과

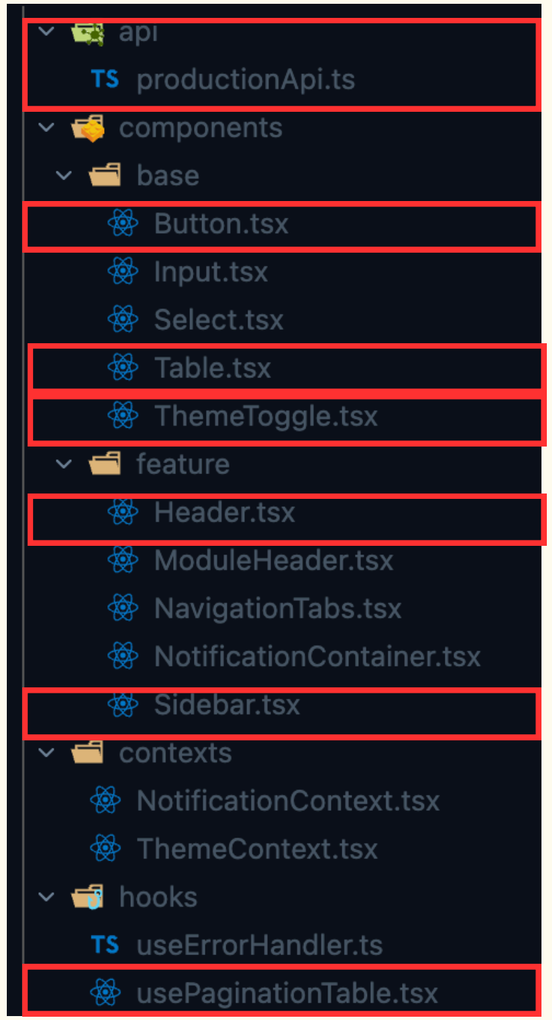
- 두 플랫폼 간 비즈니스 로직의 일관성 확보
- 유지보수와 확장성 향상
- 개발 시간 단축

FSD (Featured-Slice Design)

배경

- 페이지가 많아지면서 전통적인 방식(component, hook, page)으로 분리하는 기능분할은 낮은 응집도로 인해 확장이 어려워짐





FSD (Featured-Slice Design)

과제

○ 백엔드에서 진행하는 도메인 단위의 기능 개발에 적합한 아키텍처 구조를 선택

백엔드는 현재 모듈별로 분리된 엔드포인트들 즉, 백엔드에서도 공장, 창고가 아닌 공장을 관리, 창고를 관리 같은 비즈니스 도메인 단위로 나누어져있다.

07 개인 발표 - 추창우

FSD (Feature-Slice Design)

액션 ○ 아키텍처 비교후 학습

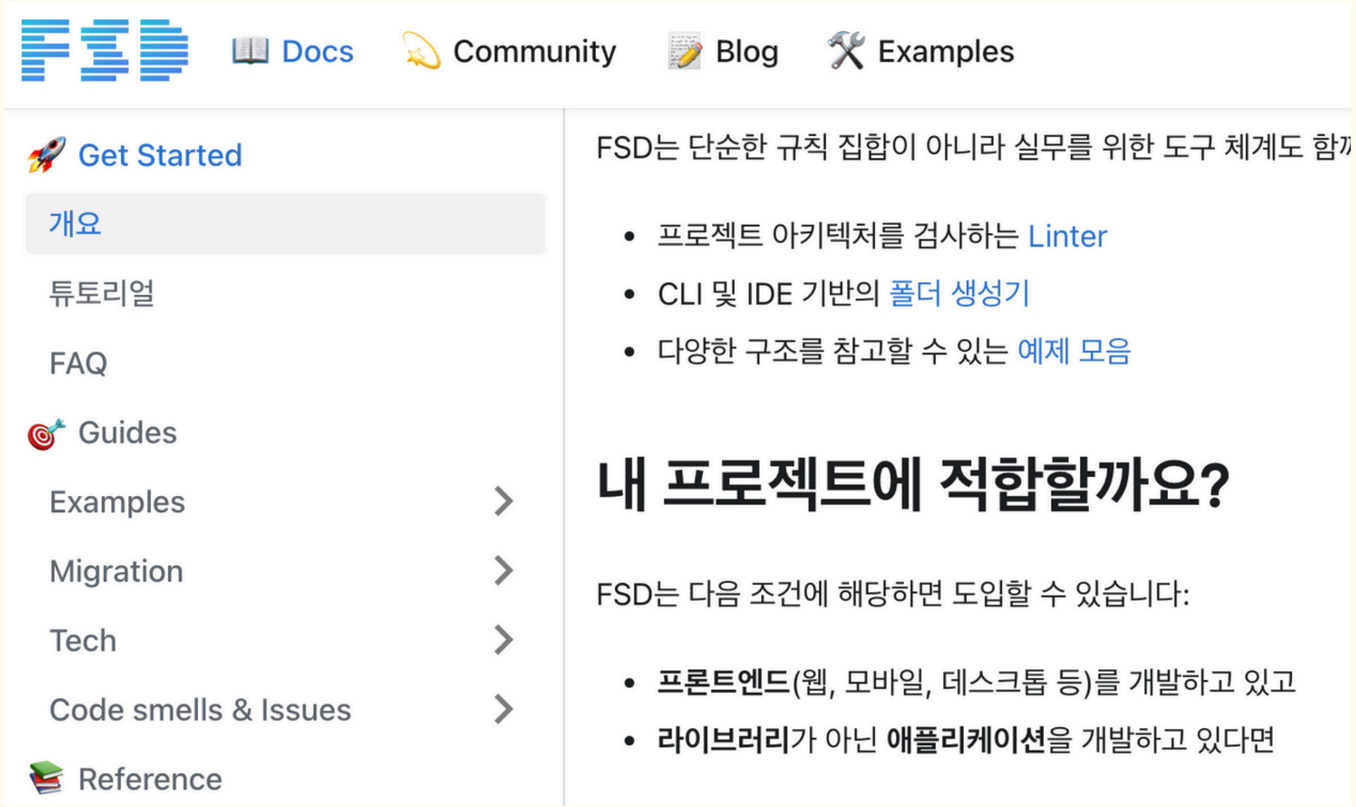
1. vs. 타입 기반 구조 (Folder-by-Type)
   - src/components, src/hooks, src/pages, src/api 등 기술 스택 유형별로 폴더를 구성
   - 'BOM 생성' 기능 하나를 수정하기 위해 components, hooks, api 폴더를 모두 수정

2. vs. 아토믹 디자인 (Atomic Design)
   - src/atoms, src/molecules, src/organisms, src/templates, src/pages로 UI 컴포넌트를 분리
   - UI 컴포넌트의 재사용성과 디자인 시스템 구축에만 초점을 맞춘다. 비즈니스 로직, API 호출, 상태 관리의 위치에 대해서는 명확히 규정하지 않음

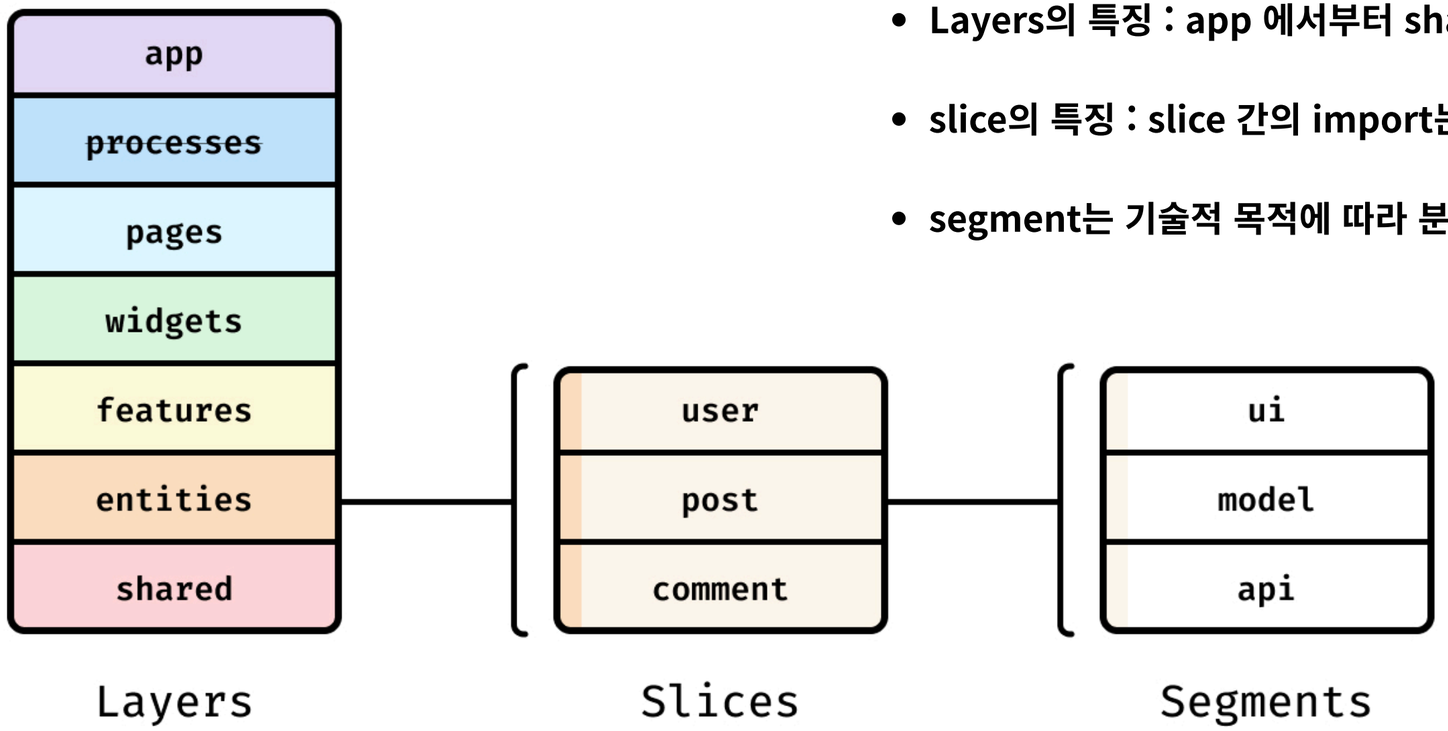
3. vs. 도메인 주도 개발(DDD)
   - 너무 추상적인 명세로 인해 초기 학습 난이도가 너무 높다고 판단

07 개인 발표 - 추창우

FSD (Feature-Slice Design)



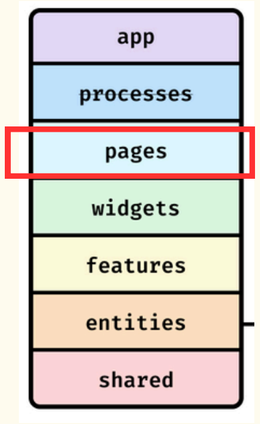
FSD (Feature-Slice Design)



- Layers의 특징 : app 에서부터 shared로 내려가는 계층 구조. 위로는 import하는 것을 금지
- slice의 특징 : slice 간의 import는 불가능
- segment는 기술적 목적에 따라 분류

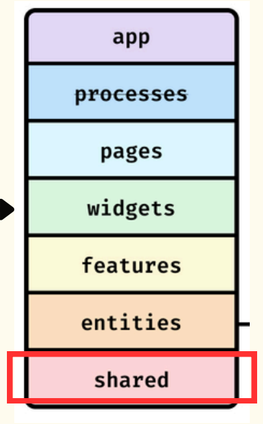
당장 나누어서 구현하기보다 Pages에서 3번 이상 겹치는 기능이 있다면 리팩토링을 하기로 결정

FSD (Feature-Slice Design)



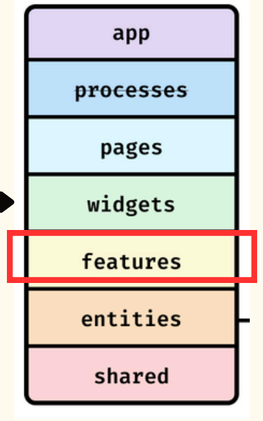
입고관리 페이지

```
<SearchFilterBar
  searchTerm={searchTerm}
  onSearchChange={setSearchTerm}
  searchPlaceholder="품목 코드, 품목명 검색..."
  filters={[
```



모든 페이지에서 사용

```
<ReceivingProcessForm
  warehouseId={Number(warehouseId)}
  processId={Number(processId)}
  onSucess={handleSuccess}
  onCancel={handleCancel}
/>
```



엔티티와 결합해 행동이 들어감

FSD (Feature-Slice Design)

높은 신뢰도가 요구되는 share슬라이스의 기본 ui는 TDD로 개발

```
// disapble 상태에서는 입력이 되면 안된다.
it('disabled 상태에서는 값을 입력할 수 없어야 한다', async () => {
  const user = userEvent.setup()

  render(<Input disabled />)

  const inputElement = screen.getByRole('textbox')
```

```
// disabled 상태에서 스타일이 올바르게 적용되어 있어야한다.
it('disabled 상태에서는 올바른 스타일이 적용되어야 한다', () => {
  render(<Input disabled />)
  const inputElement = screen.getByRole('textbox')
  expect(inputElement).toHaveAttribute('disabled')
})

// cva로 정의한 variant에 따라 다른 스타일이 적용되어야 한다.
it('variant에 따라 다른 스타일이 적용되어야 한다', () => {
  const { rerender } = render(<Input variant="primary" />)
  const inputElement = screen.getByRole('textbox')
```

FSD (Feature-Slice Design)

결과

아직도 진짜로 FSD를 활용하는지는 모르겠다
FSD의 Slice와 Segment 의존성 규칙이 개발자의 '사고방식의 흐름'을 자연스럽게 올바른 방향으로 이끄는 것을 경험

07 발표 및 소감

프로젝트 진행 중 배운 점

- 실제 개발하고자 했던 MVP 기능 및 추가 기능 모두 구현하였습니다.
- 단순히 각자 맡은 부분만 작업하는 것이 아니라, 피그마를 활용해 와이어프레임과 실제 디자인을 공유하면서 각 OS별 팀원들의 피드백을 빠르게 반영했고, 덕분에 의도와 결과물이 일치하는 경험을 할 수 있었습니다.
- 매일 아침 Scrum 회의를 통해 팀원 간의 업무 진행 상황을 공유 및 발생한 이슈를 신속하게 해결할 수 있었습니다.
- 개발 중간마다 UI에 대한 팀 내·외부의 피드백을 받아 적극 반영했고, 그 결과 프로젝트 후반으로 갈수록 화면의 완성도와 사용자 편의성이 향상되었습니다.
- 각 서비스가 자체 데이터베이스를 가지고 이벤트로 정보를 주고받는 과정에서, 서로의 책임 범위를 명확히 정하는 것이 시스템 안정성과 유지보수에 큰 영향을 준다는 것을 배웠습니다.

느낀 점 및 향후 개선 아이디어

- 예상보다 구현 난이도가 높거나, 기능 세분화가 부족하여 개발 일정 산정이 부정확했고 개발 마감 시점에 대한 부담이 컸습니다.
- 프로젝트 초기 단계에서 API 명세 등의 프로젝트 구조를 모호하게 잡아 실질적인 개발이 지연되었습니다.
- 예상치 못한 응답값(예: null, 빈 배열 등)에 대한 방어 로직이 부족해, 개발 중 일부 페이지에서 발생한 오류의 원인을 파악하는 데 시간이 소요되었습니다.
- 서비스 간 연결로 인해 한 변화가 여러 곳에 영향을 주는 경험을 통해, 코드보다 이벤트 흐름과 데이터 책임을 명확히 정하는 것이 더 중요함을 깨달았습니다.

삼삼오토

감사합니다