

21. 실전 프로젝트 설계 예시

게시판 및 댓글 시스템

📌 1. 핵심 요구사항

기능	설명
게시글 CRUD	게시글 작성, 조회, 수정, 삭제
댓글 CRUD	특정 게시글에 댓글 작성, 조회, 수정, 삭제
페이징	게시글 및 댓글 리스트는 페이지 단위로
정렬	최신순, 추천순 등 (선택적)
연관관계	게시글 1:N 댓글 구조
트랜잭션	댓글 작성 시 게시글 존재 여부 검증 포함

🏗️ 2. 도메인 모델 설계

◆ Entity 구조



- 게시글(Post)은 여러 댓글(Comment)을 가질 수 있음
- 댓글은 하나의 게시글에만 속함

◆ Post Entity

```
1  @Entity
2  public class Post {
3      @Id @GeneratedValue
4      private Long id;
5
6      private String title;
7      private String content;
8      private String writer;
9
10     @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
11     private List<Comment> comments = new ArrayList<>();
12
13     private LocalDateTime createdAt;
14     private LocalDateTime updatedAt;
15
16     @PrePersist
17     public void created() {
```

```

18         createdAt = updatedAt = LocalDateTime.now();
19     }
20
21     @PreUpdate
22     public void updated() {
23         updatedAt = LocalDateTime.now();
24     }
25 }

```

◆ Comment Entity

```

1  @Entity
2  public class Comment {
3      @Id @GeneratedValue
4      private Long id;
5
6      private String content;
7      private String writer;
8
9      @ManyToOne(fetch = FetchType.LAZY)
10     @JoinColumn(name = "post_id")
11     private Post post;
12
13     private LocalDateTime createdAt;
14
15     @PrePersist
16     public void created() {
17         createdAt = LocalDateTime.now();
18     }
19 }

```

3. DTO 계층 설계

PostDto

```

1  public class PostDto {
2      private Long id;
3      private String title;
4      private String content;
5      private String writer;
6      private LocalDateTime createdAt;
7      private int commentCount;
8  }

```

CommentDto

```
1 public class CommentDto {
2     private Long id;
3     private String content;
4     private String writer;
5     private LocalDateTime createdAt;
6 }
```

❧ 4. Repository 계층

```
1 public interface PostRepository extends JpaRepository<Post, Long> {
2 }
3
4 public interface CommentRepository extends JpaRepository<Comment, Long> {
5     List<Comment> findByPostId(Long postId);
6 }
```

🧠 5. Service 계층

```
1 @Service
2 @RequiredArgsConstructor
3 public class PostService {
4     private final PostRepository postRepository;
5
6     @Transactional
7     public Long createPost(PostDto dto) {
8         Post post = new Post();
9         post.setTitle(dto.getTitle());
10        post.setContent(dto.getContent());
11        post.setWriter(dto.getWriter());
12        return postRepository.save(post).getId();
13    }
14
15    public PostDto getPost(Long id) {
16        Post post = postRepository.findById(id).orElseThrow();
17        return mapToDto(post);
18    }
19
20    @Transactional
21    public void updatePost(Long id, PostDto dto) {
22        Post post = postRepository.findById(id).orElseThrow();
23        post.setTitle(dto.getTitle());
24        post.setContent(dto.getContent());
25    }
26
27    @Transactional
28    public void deletePost(Long id) {
29        postRepository.deleteById(id);
30    }
31 }
```

```

30     }
31
32     private PostDto mapToDto(Post post) {
33         return new PostDto(
34             post.getId(),
35             post.getTitle(),
36             post.getContent(),
37             post.getWriter(),
38             post.getCreatedAt(),
39             post.getComments().size()
40         );
41     }
42 }

```

CommentService

```

1  @Service
2  @RequiredArgsConstructor
3  public class CommentService {
4      private final CommentRepository commentRepository;
5      private final PostRepository postRepository;
6
7      @Transactional
8      public Long addComment(Long postId, CommentDto dto) {
9          Post post = postRepository.findById(postId).orElseThrow();
10         Comment comment = new Comment();
11         comment.setContent(dto.getContent());
12         comment.setWriter(dto.getWriter());
13         comment.setPost(post);
14         return commentRepository.save(comment).getId();
15     }
16
17     public List<CommentDto> getComments(Long postId) {
18         return commentRepository.findByPostId(postId)
19             .stream()
20             .map(this::mapToDto)
21             .toList();
22     }
23
24     @Transactional
25     public void deleteComment(Long commentId) {
26         commentRepository.deleteById(commentId);
27     }
28
29     private CommentDto mapToDto(Comment comment) {
30         return new CommentDto(
31             comment.getId(),
32             comment.getContent(),
33             comment.getWriter(),
34             comment.getCreatedAt()
35         );

```

```
36     }
37 }
```

6. Controller 계층

```
1  @RestController
2  @RequestMapping("/api/posts")
3  @RequiredArgsConstructor
4  public class PostController {
5      private final PostService postService;
6
7      @PostMapping
8      public ResponseEntity<Long> createPost(@RequestBody PostDto dto) {
9          return ResponseEntity.ok(postService.createPost(dto));
10     }
11
12     @GetMapping("/{id}")
13     public ResponseEntity<PostDto> getPost(@PathVariable Long id) {
14         return ResponseEntity.ok(postService.getPost(id));
15     }
16
17     @PutMapping("/{id}")
18     public void updatePost(@PathVariable Long id, @RequestBody PostDto dto) {
19         postService.updatePost(id, dto);
20     }
21
22     @DeleteMapping("/{id}")
23     public void deletePost(@PathVariable Long id) {
24         postService.deletePost(id);
25     }
26 }
```

```
1  @RestController
2  @RequestMapping("/api/posts/{postId}/comments")
3  @RequiredArgsConstructor
4  public class CommentController {
5      private final CommentService commentService;
6
7      @PostMapping
8      public ResponseEntity<Long> addComment(@PathVariable Long postId, @RequestBody
CommentDto dto) {
9          return ResponseEntity.ok(commentService.addComment(postId, dto));
10     }
11
12     @GetMapping
13     public ResponseEntity<List<CommentDto>> getComments(@PathVariable Long postId) {
14         return ResponseEntity.ok(commentService.getComments(postId));
15     }
16
17     @DeleteMapping("/{commentId}")
18     public void deleteComment(@PathVariable Long commentId) {
```

```
19 |         commentService.deleteComment(commentId);
20 |     }
21 | }
```

7. 확장 포인트

기능	확장 방향
조회수 기능	@Version 또는 별도 viewCount 필드
좋아요 기능	PostLike 엔티티 추가
대댓글 기능	Comment 에 parentId 필드 추가
Soft Delete	deleted 플래그 추가
페이징/정렬	Pageable 활용 (Spring Data JPA)
검색 기능	PostRepository 에 findByTitleContaining 등 추가

마무리 요약

구성 요소	핵심 내용
엔티티	Post , Comment - 1:N 관계
서비스	트랜잭션 단위로 작성, DTO 매핑 포함
컨트롤러	RESTful 방식으로 설계
연관관계	@OneToMany , @ManyToOne 사용
확장성	좋아요, 대댓글, 조회수 등 구조적 확장 가능

쇼핑몰 주문 시스템

1. 주요 요구사항

기능	설명
상품 조회	카탈로그, 상세 보기
장바구니	담기, 수정, 삭제, 조회
주문 생성	장바구니 기반 주문 생성
결제 처리	PG 연동 또는 모의 결제
주문 조회	내 주문 목록, 상세 확인

기능	설명
재고 차감	주문 완료 시 상품 재고 감소
취소 처리	주문 취소 및 재고 복구

2. 도메인 모델 설계 (Entity)

```
1 Member
2 Product ---< Stock
3       \---< CartItem
4       \---< OrderItem ---< Order
```

◆ Member (회원)

```
1 @Entity
2 public class Member {
3     @Id @GeneratedValue
4     private Long id;
5     private String username;
6     private String email;
7     private String password;
8 }
```

◆ Product (상품)

```
1 @Entity
2 public class Product {
3     @Id @GeneratedValue
4     private Long id;
5     private String name;
6     private int price;
7     private String description;
8
9     private int stockQuantity;
10
11     public void decreaseStock(int quantity) {
12         if (this.stockQuantity < quantity)
13             throw new IllegalStateException("재고 부족");
14         this.stockQuantity -= quantity;
15     }
16
17     public void increaseStock(int quantity) {
18         this.stockQuantity += quantity;
19     }
20 }
```

◆ CartItem (장바구니 항목)

```
1  @Entity
2  public class CartItem {
3      @Id @GeneratedValue
4      private Long id;
5
6      @ManyToOne(fetch = FetchType.LAZY)
7      private Member member;
8
9      @ManyToOne(fetch = FetchType.LAZY)
10     private Product product;
11
12     private int quantity;
13 }
```

◆ Order / OrderItem (주문)

```
1  @Entity
2  public class Order {
3      @Id @GeneratedValue
4      private Long id;
5
6      @ManyToOne(fetch = FetchType.LAZY)
7      private Member member;
8
9      @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
10     private List<OrderItem> items = new ArrayList<>();
11
12     private LocalDateTime orderDate;
13     private OrderStatus status;
14
15     public static Order createOrder(Member member, List<OrderItem> items) {
16         Order order = new Order();
17         order.member = member;
18         order.orderDate = LocalDateTime.now();
19         order.status = OrderStatus.ORDERED;
20         for (OrderItem item : items) {
21             order.items.add(item);
22             item.setOrder(order);
23         }
24         return order;
25     }
26
27     public void cancel() {
28         this.status = OrderStatus.CANCELLED;
29         for (OrderItem item : items)
30             item.cancel();
31     }
32 }
33
```



```

34 @Entity
35 public class OrderItem {
36     @Id @GeneratedValue
37     private Long id;
38
39     @ManyToOne(fetch = FetchType.LAZY)
40     private Order order;
41
42     @ManyToOne(fetch = FetchType.LAZY)
43     private Product product;
44
45     private int orderPrice;
46     private int quantity;
47
48     public void cancel() {
49         product.increaseStock(quantity);
50     }
51 }

```

3. 서비스 계층

◆ CartService

```

1  @Service
2  @RequiredArgsConstructor
3  public class CartService {
4      private final CartItemRepository cartItemRepository;
5      private final ProductRepository productRepository;
6      private final MemberRepository memberRepository;
7
8      @Transactional
9      public void addToCart(Long memberId, Long productId, int quantity) {
10         Member member = memberRepository.findById(memberId).orElseThrow();
11         Product product = productRepository.findById(productId).orElseThrow();
12
13         CartItem item = cartItemRepository.findByMemberAndProduct(member, product)
14             .orElseGet(() -> new CartItem(member, product, 0));
15         item.setQuantity(item.getQuantity() + quantity);
16         cartItemRepository.save(item);
17     }
18
19     public List<CartItemDto> getCart(Long memberId) {
20         return cartItemRepository.findByMemberId(memberId)
21             .stream()
22             .map(CartItemDto::from)
23             .toList();
24     }
25
26     @Transactional
27     public void removeFromCart(Long itemId) {
28         cartItemRepository.deleteById(itemId);

```

```
29     }
30 }
```

◆ OrderService

```
1  @Service
2  @RequiredArgsConstructor
3  public class OrderService {
4      private final MemberRepository memberRepository;
5      private final CartItemRepository cartItemRepository;
6      private final OrderRepository orderRepository;
7
8      @Transactional
9      public Long createOrder(Long memberId) {
10         Member member = memberRepository.findById(memberId).orElseThrow();
11         List<CartItem> cartItems = cartItemRepository.findByMemberId(memberId);
12         if (cartItems.isEmpty()) throw new IllegalStateException("장바구니 비어있음");
13
14         List<OrderItem> orderItems = cartItems.stream().map(item -> {
15             Product product = item.getProduct();
16             product.decreaseStock(item.getQuantity());
17             return new OrderItem(product, item.getQuantity(), product.getPrice());
18         }).toList();
19
20         Order order = Order.createOrder(member, orderItems);
21         cartItemRepository.deleteAll(cartItems);
22         return orderRepository.save(order).getId();
23     }
24
25     public OrderDto getOrder(Long orderId) {
26         return OrderDto.from(orderRepository.findById(orderId).orElseThrow());
27     }
28
29     @Transactional
30     public void cancelOrder(Long orderId) {
31         Order order = orderRepository.findById(orderId).orElseThrow();
32         order.cancel();
33     }
34 }
```

4. API Controller 구조

```
1  @RestController
2  @RequestMapping("/api/cart")
3  @RequiredArgsConstructor
4  public class CartController {
5      private final CartService cartService;
6
7      @PostMapping
```

```

8      public void addToCart(@RequestBody CartRequest req) {
9          cartService.addToCart(req.memberId(), req.productId(), req.quantity());
10     }
11
12     @GetMapping("/{memberId}")
13     public List<CartItemDto> getCart(@PathVariable Long memberId) {
14         return cartService.getCart(memberId);
15     }
16
17     @DeleteMapping("/{itemId}")
18     public void remove(@PathVariable Long itemId) {
19         cartService.removeFromCart(itemId);
20     }
21 }

```

```

1  @RestController
2  @RequestMapping("/api/orders")
3  @RequiredArgsConstructor
4  public class OrderController {
5      private final OrderService orderService;
6
7      @PostMapping("/{memberId}")
8      public Long createOrder(@PathVariable Long memberId) {
9          return orderService.createOrder(memberId);
10     }
11
12     @GetMapping("/{orderId}")
13     public OrderDto getOrder(@PathVariable Long orderId) {
14         return orderService.getOrder(orderId);
15     }
16
17     @PostMapping("/cancel/{orderId}")
18     public void cancelOrder(@PathVariable Long orderId) {
19         orderService.cancelOrder(orderId);
20     }
21 }

```

5. 트랜잭션 및 예외 처리 전략

항목	전략
주문 생성	<code>@Transactional</code> 전체 묶음, 재고 차감 포함
주문 취소	<code>@Transactional</code> 로 상태 변경 + 재고 복구
예외 상황	재고 부족 → <code>IllegalStateException</code> , 404/400 처리



6. 확장 포인트

기능	확장
결제 연동	PG사 API 연동 후 결제 성공 → 주문 저장 순으로
쿠폰	Coupon, CouponUse, Order.couponDiscount
배송	Delivery 엔티티 추가, 상태값 READY, SHIPPED 등
주문 상태별 분기	ORDERED, PAID, SHIPPED, CANCELLED
이벤트 처리	주문 완료 → Kafka 알림, 이메일 발송 등



마무리 요약

구성 요소	내용
도메인	Member, Product, CartItem, Order, OrderItem
주요 기능	장바구니 관리, 주문 생성, 재고 차감, 주문 취소
트랜잭션	주문 생성/취소는 단일 트랜잭션 보장
확장성	결제, 배송, 이벤트 시스템 등 연계 가능
패턴	DDD 스타일, 서비스 계층 중심 구조

다음 주제

- 📦 주문 상태별 흐름도/시나리오 설계
- 💳 결제 모듈 연동 (KG이니시스, TossPayments 등)
- 🌐 프론트엔드 연결 (React + REST)
- 🧪 단위 테스트 + 통합 테스트 구조 설계
- 📬 CQRS / 이벤트 기반 주문 처리 (Kafka)

회원가입 및 JWT 인증 시스템



1. 주요 기능 요구사항

기능	설명
회원가입	이메일/비밀번호/닉네임 등록, 비밀번호 암호화
로그인	로그인 성공 시 JWT 토큰 발급
JWT 인증	토큰으로 유저 인증, 세션 없음

기능	설명
인증 필터	요청마다 토큰 검증 및 인증 처리
인가 처리	특정 API는 로그인한 사용자만 접근 가능

2. Entity 및 DTO

◆ User Entity

```
1  @Entity
2  @Table(name = "users")
3  public class User {
4      @Id @GeneratedValue
5      private Long id;
6
7      @Column(unique = true)
8      private String email;
9
10     private String password;
11     private String nickname;
12
13     private String role = "USER"; // 기본 역할
14 }
```

◆ DTO 정의

```
1  public record SignupRequest(String email, String password, String nickname) {}
2
3  public record LoginRequest(String email, String password) {}
4
5  public record TokenResponse(String accessToken) {}
```

3. 비밀번호 암호화 및 회원가입

◆ BCrypt 암호화

```
1  @Bean
2  public PasswordEncoder passwordEncoder() {
3      return new BCryptPasswordEncoder();
4  }
```

◆ 회원가입 서비스

```
1 @Service
2 @RequiredArgsConstructor
3 public class AuthService {
4     private final UserRepository userRepository;
5     private final PasswordEncoder passwordEncoder;
6
7     @Transactional
8     public void signup(SignupRequest request) {
9         if (userRepository.existsByEmail(request.email())) {
10             throw new IllegalArgumentException("이미 가입된 이메일입니다.");
11         }
12
13         User user = new User();
14         user.setEmail(request.email());
15         user.setNickname(request.nickname());
16         user.setPassword(passwordEncoder.encode(request.password()));
17         userRepository.save(user);
18     }
19 }
```

4. JWT 토큰 발급

◆ JWT 의존성 (Gradle)

```
1 implementation 'io.jsonwebtoken:jjwt-api:0.11.5'
2 runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5'
3 runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.11.5'
```

◆ JwtTokenProvider

```
1 @Component
2 public class JwtTokenProvider {
3     @Value("${jwt.secret}")
4     private String secret;
5
6     private final long EXPIRATION = 1000L * 60 * 60;
7
8     public String generateToken(String email) {
9         Date now = new Date();
10         Date expiry = new Date(now.getTime() + EXPIRATION);
11
12         return Jwts.builder()
13             .setSubject(email)
14             .setIssuedAt(now)
15             .setExpiration(expiry)
16             .signWith(Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8)))
17             .compact();
18     }
19 }
```

```

18     }
19
20     public String getEmailFromToken(String token) {
21         return Jwts.parserBuilder()
22             .setSigningKey(secret.getBytes(StandardCharsets.UTF_8))
23             .build()
24             .parseClaimsJws(token)
25             .getBody()
26             .getSubject();
27     }
28
29     public boolean validate(String token) {
30         try {
31             getEmailFromToken(token);
32             return true;
33         } catch (Exception e) {
34             return false;
35         }
36     }
37 }

```

5. 로그인 및 토큰 응답

```

1  @RequiredArgsConstructor
2  @RestController
3  @RequestMapping("/api/auth")
4  public class AuthController {
5      private final AuthService authService;
6      private final UserRepository userRepository;
7      private final PasswordEncoder passwordEncoder;
8      private final JwtTokenProvider jwtProvider;
9
10     @PostMapping("/signup")
11     public ResponseEntity<Void> signup(@RequestBody SignupRequest request) {
12         authService.signup(request);
13         return ResponseEntity.ok().build();
14     }
15
16     @PostMapping("/login")
17     public ResponseEntity<TokenResponse> login(@RequestBody LoginRequest request) {
18         User user = userRepository.findByEmail(request.email())
19             .orElseThrow(() -> new UsernameNotFoundException("이메일 없음"));
20
21         if (!passwordEncoder.matches(request.password(), user.getPassword())) {
22             throw new BadCredentialsException("비밀번호 불일치");
23         }
24
25         String token = jwtProvider.generateToken(user.getEmail());
26         return ResponseEntity.ok(new TokenResponse(token));
27     }
28 }

```

6. JWT 인증 필터

```
1 public class JwtAuthenticationFilter extends OncePerRequestFilter {
2     private final JwtTokenProvider jwtProvider;
3     private final UserRepository userRepository;
4
5     public JwtAuthenticationFilter(JwtTokenProvider jwtProvider, UserRepository
userRepository) {
6         this.jwtProvider = jwtProvider;
7         this.userRepository = userRepository;
8     }
9
10    @Override
11    protected void doFilterInternal(HttpServletRequest request,
12                                    HttpServletResponse response,
13                                    FilterChain chain) throws ServletException,
IOException {
14
15        String header = request.getHeader("Authorization");
16        if (header != null && header.startsWith("Bearer ")) {
17            String token = header.substring(7);
18            if (jwtProvider.validate(token)) {
19                String email = jwtProvider.getEmailFromToken(token);
20                User user = userRepository.findByEmail(email).orElseThrow();
21                UsernamePasswordAuthenticationToken auth =
22                    new UsernamePasswordAuthenticationToken(user, null,
23                                                            List.of(new SimpleGrantedAuthority(user.getRole())));
24                SecurityContextHolder.getContext().setAuthentication(auth);
25            }
26        }
27
28        chain.doFilter(request, response);
29    }
30 }
```

7. Spring Security 설정

```
1 @Configuration
2 @EnableWebSecurity
3 @RequiredArgsConstructor
4 public class SecurityConfig {
5     private final JwtTokenProvider jwtProvider;
6     private final UserRepository userRepository;
7
8     @Bean
9     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
10         return http
11             .csrf(AbstractHttpConfigurer::disable)
```



```

12         .sessionManagement(session ->
13             session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
14             .authorizeHttpRequests(auth -> auth
15                 .requestMatchers("/api/auth/**").permitAll()
16                 .anyRequest().authenticated()
17             )
18             .addFilterBefore(new JwtAuthenticationFilter(jwtProvider,
19                 userRepository),
20                 UsernamePasswordAuthenticationFilter.class)
21             .build();
22     }
23 }

```

✓ 8. 실행 흐름 요약

```

1  [회원가입]
2    - 비밀번호 암호화 후 DB 저장
3
4  [로그인]
5    - 이메일/비밀번호 검증 → JWT 발급
6
7  [인증 요청]
8    - Authorization: Bearer <token> 헤더 포함
9    - JwtAuthenticationFilter가 인증 처리
10   - SecurityContext에 유저 정보 저장됨

```

🔑 9. application.yml 설정 예시

```





1  jwt:
2    secret: MY_SECRET_12345678901234567890ABCDEF

```

✓ 마무리 요약

기능	구성 요소
회원가입	DTO + Service + BCryptPasswordEncoder
로그인	이메일/비밀번호 확인 후 JWT 발급
토큰 발급	<code>JwtTokenProvider.generateToken()</code>
인증 필터	<code>JwtAuthenticationFilter</code>
Spring Security 연동	Stateless, 필터 삽입, 권한 처리
확장성	Refresh Token, Role/권한별 인증 처리 가능

다음 주제

-  Refresh Token + Redis 기반 토큰 재발급
-  Role 기반 인가 (`@PreAuthorize`, `hasRole`)
-  토큰 무효화 및 블랙리스트 처리 전략
-  OAuth2 로그인 (Google, Kakao 연동)

Kafka 기반 알림 시스템

1. 요구사항 및 시나리오

시나리오	설명
이벤트 트리거	주문 완료 시 Kafka로 이벤트 발행
소비자(Consumer)	알림 서비스가 해당 이벤트를 구독하고 처리
알림 방식	이메일 / SMS / 푸시 등 확장 가능
비동기 처리	주문 흐름과 별도로 이벤트 처리
보장 요구	최소 1회 전송 (at-least-once), 로그 기반 재처리 가능

2. Kafka 기본 구성

토픽 설계

Topic 이름	메시지 타입
<code>order.notification</code>	주문 알림 이벤트

3. 알림 이벤트 DTO

```
1 public record NotificationEvent(  
2     Long userId,  
3     String type,           // EMAIL, SMS, PUSH 등  
4     String title,  
5     String content  
6 ) {}
```

Kafka는 기본적으로 JSON 직렬화로 주고받기 때문에 **DTO는 직렬화 가능해야 함**.

⚙️ 4. Kafka 설정 (application.yml)

```
1 spring:
2   kafka:
3     bootstrap-servers: localhost:9092
4     producer:
5       key-serializer: org.apache.kafka.common.serialization.StringSerializer
6       value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
7     consumer:
8       key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
9       value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
10    group-id: notification-group
11    properties:
12      spring.json.trusted.packages: "*"

```

✉️ 5. 메시지 발행 Producer

```
1 @Service
2 @RequiredArgsConstructor
3 public class NotificationProducer {
4     private final KafkaTemplate<String, NotificationEvent> kafkaTemplate;
5
6     public void send(NotificationEvent event) {
7         kafkaTemplate.send("order.notification", event);
8     }
9 }

```

🛒 6. 주문 서비스에서 발행

```
1 @Service
2 @RequiredArgsConstructor
3 public class OrderService {
4     private final NotificationProducer producer;
5     private final OrderRepository orderRepository;
6
7     @Transactional
8     public void completeOrder(Long orderId) {
9         Order order = orderRepository.findById(orderId).orElseThrow();
10        order.markCompleted();
11
12        NotificationEvent event = new NotificationEvent(
13            order.getUserId(),
14            "EMAIL",
15            "주문이 완료되었습니다.",
16            "주문 번호 " + order.getId() + " 가 성공적으로 처리되었습니다."
17        );
18
19        producer.send(event);

```

```
20     }
21 }
```

7. 알림 소비자 Consumer

```
1  @Component
2  public class NotificationConsumer {
3
4      @KafkaListener(topics = "order.notification", groupId = "notification-group")
5      public void listen(NotificationEvent event) {
6          System.out.println("✅ 알림 수신: " + event);
7
8          switch (event.type()) {
9              case "EMAIL" -> sendEmail(event);
10             case "SMS" -> sendSms(event);
11             case "PUSH" -> sendPush(event);
12         }
13     }
14
15     private void sendEmail(NotificationEvent event) {
16         // 실제 이메일 전송 로직
17         System.out.printf("📧 이메일 전송: [%s] %s\n", event.title(), event.content());
18     }
19
20     private void sendSms(NotificationEvent event) {
21         // SMS 전송 로직
22     }
23
24     private void sendPush(NotificationEvent event) {
25         // 푸시 알림 로직
26     }
27 }
```

8. 테스트

```
1  # 토픽 생성
2  kafka-topics.sh --create --topic order.notification --bootstrap-server localhost:9092
3
4  # 메시지 수신 테스트
5  kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic order.notification --from-beginning
```



9. 실전 확장 구조



구조 예시

```
1 [OrderService]
2   ↓ Kafka 메시지 발행
3 [Kafka Broker (order.notification)]
4   ↓
5 [NotificationConsumer]
6   → EmailService
7   → SmsService
8   → PushService
```



10. 확장 포인트

기능	설명
Dead Letter Queue (DLQ)	소비 실패한 메시지를 별도 토픽으로 분리
메시지 재처리	DLQ 소비자를 만들어 재시도 가능
메시지 헤더 사용	<code>KafkaHeader</code> 로 타입/우선순위 구분 가능
트레이싱	Sleuth + Kafka Propagation 연동
인증된 사용자 전파	JWT 토큰을 Kafka 메시지 헤더에 포함 가능



마무리 요약

구성 요소	설명
<code>NotificationEvent</code>	이벤트 메시지 DTO
<code>NotificationProducer</code>	Kafka 발행자
<code>NotificationConsumer</code>	Kafka 소비자 (알림 처리 담당)
<code>kafkaTemplate</code> / <code>@KafkaListener</code>	Spring Kafka 핵심 컴포넌트
<code>order.notification</code>	알림용 토픽
확장 방향	이메일 → SMS → 푸시 → 멀티 채널 처리까지 가능

다음 주제

- 📦 Kafka + Redis 기반 **중복 알림 방지** 전략
- 💬 Kafka + Retry + DLQ 패턴
- 📄 Kafka 메시지 추적 로깅 + Elastic Stack 연동
- 🌐 Kafka + Kubernetes + Externalized Secrets 연동

파일 업로드 및 썸네일 처리 시스템

📌 1. 주요 기능 요약

기능	설명
파일 업로드	이미지 또는 일반 파일 업로드 (Multipart 지원)
저장 경로 분리	업로드 파일과 썸네일 디렉터리 구분
썸네일 생성	이미지 업로드 시 자동으로 썸네일 생성
접근 경로 제공	업로드된 파일을 URL로 접근 가능하게 제공
파일 삭제	원본 및 썸네일 동시 삭제 가능

📦 2. 의존성 설정 (Gradle 기준)

```
1 dependencies {
2     implementation 'org.springframework.boot:spring-boot-starter-web'
3     implementation 'net.coobird:thumbnailator:0.4.14'
4 }
```

`thumbnailator`는 이미지 리사이징을 위한 경량 라이브러리야.

⚙️ 3. application.yml 설정

```
1 file:
2   upload-dir: uploads/
3   thumbnail-dir: uploads/thumbnails/
```

📁 4. 파일 저장 유틸리티

```
1 @Component
2 @RequiredArgsConstructor
3 public class FileStorageUtil {
4
5     @Value("${file.upload-dir}")
6     private String uploadDir;
```

```

7
8     @Value("${file.thumbnail-dir}")
9     private String thumbnailDir;
10
11     public String saveFile(MultipartFile file) throws IOException {
12         String filename = UUID.randomUUID() + "_" + file.getOriginalFilename();
13         Path targetPath = Paths.get(uploadDir).resolve(filename);
14         Files.createDirectories(targetPath.getParent());
15         file.transferTo(targetPath.toFile());
16         return filename;
17     }
18
19     public void createThumbnail(String filename, int width, int height) throws
IOException {
20         Path source = Paths.get(uploadDir).resolve(filename);
21         Path thumbTarget = Paths.get(thumbnailDir).resolve(filename);
22         Files.createDirectories(thumbTarget.getParent());
23
24         try (InputStream in = Files.newInputStream(source);
25             OutputStream out = Files.newOutputStream(thumbTarget)) {
26             Thumbnails.of(in)
27                 .size(width, height)
28                 .toOutputStream(out);
29         }
30     }
31
32     public Resource load(String filename, boolean thumbnail) {
33         try {
34             Path dir = thumbnail ? Paths.get(thumbnailDir) : Paths.get(uploadDir);
35             Path path = dir.resolve(filename);
36             return new UrlResource(path.toUri());
37         } catch (Exception e) {
38             throw new RuntimeException("파일 불러오기 실패", e);
39         }
40     }
41 }

```

5. 파일 업로드 API

```

1  @RestController
2  @RequestMapping("/api/files")
3  @RequiredArgsConstructor
4  public class FileUploadController {
5
6      private final FileStorageUtil fileStorageUtil;
7
8      @PostMapping("/upload")
9      public ResponseEntity<Map<String, String>> upload(@RequestParam("file")
MultipartFile file) throws IOException {
10         String filename = fileStorageUtil.saveFile(file);
11         if (file.getContentType().startsWith("image")) {

```

```

12         fileStorageUtil.createThumbnail(filename, 200, 200);
13     }
14
15     Map<String, String> result = new HashMap<>();
16     result.put("fileName", filename);
17     result.put("fileUrl", "/api/files/view/" + filename);
18     result.put("thumbnailUrl", "/api/files/thumb/" + filename);
19
20     return ResponseEntity.ok(result);
21 }
22
23 @GetMapping("/view/{filename}")
24 public ResponseEntity<Resource> view(@PathVariable String filename) {
25     Resource file = fileStorageUtil.load(filename, false);
26     return ResponseEntity.ok()
27         .header(HttpHeaders.CONTENT_DISPOSITION, "inline; filename=\"" +
filename + "\"")
28         .body(file);
29 }
30
31 @GetMapping("/thumb/{filename}")
32 public ResponseEntity<Resource> thumbnail(@PathVariable String filename) {
33     Resource file = fileStorageUtil.load(filename, true);
34     return ResponseEntity.ok()
35         .header(HttpHeaders.CONTENT_DISPOSITION, "inline; filename=\"" +
filename + "\"")
36         .body(file);
37 }
38 }

```

6. 확장 가능 기능

기능	설명
파일 확장자 필터링	<code>.jpg</code> , <code>.png</code> , <code>.pdf</code> 등 제한 가능
MIME Type 검증	<code>file.getContentType()</code> 로 허용 여부 확인
썸네일 크기 설정	요청마다 동적 지정 가능하게 확장
DB 연동	업로드 파일 메타정보를 DB에 저장 (경로, 사용자, 생성일 등)
AWS S3 연동	파일 저장소를 로컬 → S3로 확장 가능
인증 연동	<code>@PreAuthorize("isAuthenticated()")</code> 등으로 권한 설정

✅ 마무리 요약

기능	구현 방식
파일 저장	UUID 기반 파일명 생성 + <code>MultipartFile.transferTo()</code>
썸네일 생성	<code>Thumbnailator.of().size().toOutputStream()</code>
접근 API	<code>/api/files/view/{filename}</code> , <code>/thumb/{filename}</code>
에러 처리	파일 존재하지 않을 경우 <code>RuntimeException</code> 발생
정리 포인트	원본/썸네일 폴더 분리, 파일명 충돌 방지, 리소스 직렬화

다음 주제

- 🗄️ 업로드 메타정보 DB 연동 (`FileMetadata`, 사용자 ID, 업로드 일시 등)
- 🌐 AWS S3 / Cloud Storage 연동
- 🔒 로그인 사용자의 개인 파일 분리 (`/user/{id}/files`)
- 🎥 동영상 썸네일 추출 (ffmpeg 연동)

마이크로서비스 기반 도서 API 시스템

📌 1. 핵심 아키텍처 개요

✂️ 서비스 분리 구조 (MSA)

서비스 이름	책임 도메인
Book Service	도서 정보 등록/조회
User Service	사용자 등록, 로그인
Review Service	도서 리뷰 등록 및 조회
Search Service	도서 검색 (Elasticsearch 기반 가능)
API Gateway	클라이언트 진입점, 라우팅
Config Server	설정 정보 중앙 관리
Discovery Server	서비스 레지스트리 (Eureka)

2. 각 서비스 개요

Book Service

- `/books`
 - GET `/books` : 전체 목록
 - GET `/books/{id}` : 단건 조회
 - POST `/books` : 등록
- DB: MySQL (책 테이블)

User Service

- `/users`
 - POST `/users` : 회원가입
 - POST `/login` : 로그인 (JWT 발급)
- JWT 인증 필터 포함
- DB: MySQL (user 테이블)

Review Service

- `/reviews`
 - POST `/reviews` : 리뷰 작성
 - GET `/reviews/book/{bookId}` : 도서 리뷰 조회
- 도서 ID 기반 리뷰 연동
- DB: MongoDB 또는 MySQL

Search Service (선택)

- `/search`
 - GET `/search?q=자바`
- Elasticsearch 연동 또는 DB LIKE 검색

3. API Gateway (Spring Cloud Gateway)

application.yml

```
1 spring:
2   cloud:
3     gateway:
4       routes:
5         - id: book-service
6           uri: lb://BOOK-SERVICE
7           predicates:
8             - Path=/books/**
9         - id: user-service
```

```

10      uri: lb://USER-SERVICE
11      predicates:
12        - Path=/users/**,/login
13    - id: review-service
14      uri: lb://REVIEW-SERVICE
15      predicates:
16        - Path=/reviews/**

```

`lb://`는 Eureka 등록된 서비스 이름 기반으로 라우팅

4. Config Server (Spring Cloud Config)

- 모든 서비스 설정을 Git 또는 로컬에 저장
- `application.yml` 파일 분리 관리

```

1  config-repo/
2  ├── book-service.yml
3  ├── user-service.yml
4  └── review-service.yml

```

각 서비스는 `spring.cloud.config.uri` 로 설정 서버를 참조함

5. Discovery Server (Eureka)

✓ 설정

```

1  eureka:
2    client:
3      register-with-eureka: true
4      fetch-registry: true
5    instance:
6      prefer-ip-address: true

```

모든 서비스는 Eureka Client로 등록되고, Gateway가 이를 참조해 라우팅

6. JWT 인증 연동

구조

```

1  [클라이언트]
2    ↓ 로그인 (User Service)
3  [JWT 발급]
4    ↓ Authorization: Bearer <token>
5  [Gateway]
6    ↓ 인증 필터 → User 정보 추출
7  [Backend Service] ← 인증된 사용자 정보 전달

```

- Gateway에서 토큰 검증 후 헤더에 사용자 ID 추가 (x-user-id)
- 후방 서비스에서는 별도 인증 없이 헤더 기반 사용자 확인

7. 도서 + 리뷰 연동 흐름 예시

1. /books/123 → BookService가 도서 정보 응답
2. /reviews/book/123 → ReviewService가 리뷰 리스트 응답
3. 클라이언트에서 정보 결합, 또는 API Gateway에서 Fan-out 패턴 사용해 한 번에 응답 구성 가능

8. 기술 스택 제안

영역	기술
서비스 개발	Spring Boot 3.x
API 게이트웨이	Spring Cloud Gateway
서비스 레지스트리	Spring Cloud Eureka
설정 관리	Spring Cloud Config
데이터 저장소	MySQL (책/유저), MongoDB (리뷰)
인증	JWT + Gateway Filter
메시징 (확장)	Kafka (ex: 리뷰 작성 시 알림 발행)

9. CI/CD 및 배포

단계	도구
코드 저장소	Git + GitHub/GitLab
CI	GitHub Actions / Jenkins
빌드	Gradle + Docker
배포	Docker Compose / Kubernetes (확장 시)
관측	Prometheus + Grafana, ELK 또는 Loki

마무리 요약

구성 요소	설명
서비스 분리	Book, User, Review 등 각 도메인으로 분리

구성 요소	설명
통신 방식	REST + Eureka + Gateway
인증 방식	JWT, Gateway 필터에서 검증
설정 방식	Config Server를 통해 중앙화
확장성	메시징(Kafka), 검색(Elastic), 알림 시스템 등 연동 가능

다음 주제

- 📡 서비스 간 통신: REST vs Kafka 이벤트 기반
- 🔑 Gateway에서 JWT 인증 + 역할(Role) 인가 처리
- ☁️ Kubernetes 기반 배포 구성
- 📄 Saga 패턴 기반 주문-결제 처리 (MSA 트랜잭션)