Phase 2 (Next): Inventory + Order lifecycle

🔥 Step 1: Rename Quantity → Stock (Clear Meaning)

In e-commerce, quantity is confusing.
Better name: stock.

So change:

```
@NotNull(message = "Product quantity cannot be null.")
@Min(value = 1, message = "Product quantity must be at least 1.")
@Column(nullable = false)
private Integer quantity;
```

👉 To:

```
@NotNull(message = "Product stock cannot be null.")
@Min(value = 0, message = "Product stock cannot be negative.")
@Column(nullable = false)
private Integer stock;
```

Why?

- Stock can be 0 (out of stock)
- Quantity usually means order quantity

This improves domain clarity.

🔥 Step 2: Add Optimistic Locking (VERY IMPORTANT)

Add this field:

```
@Version
private Long version;
```

Your final updated entity should look like:

```
@Version
private Long version;

@NotNull(message = "Product stock cannot be null.")
@Min(value = 0, message = "Product stock cannot be negative.")
@Column(nullable = false)
private Integer stock;
```

This enables optimistic locking automatically.

Now if two users try to update the same product row simultaneously, one will fail safely.

🔥 **Step 3: Database Change**

If you already have data, run:

ALTER TABLE products ADD COLUMN version BIGINT DEFAULT 0;

And if renaming:

ALTER TABLE products RENAME COLUMN quantity TO stock;

🔥 **Step 4: Now change in reduceStock in product-service we need to add @Transactional**

```
@Override
@Transactional
public void reduceStock(Long productId, Integer quantity) {

    if (quantity <= 0) {
        throw new RuntimeException("Quantity must be greater than 0");
    }

    Product product = productRepository.findById(productId)
            .orElseThrow(() ->
                    new ResourceNotFoundException("Product not found with id: " + productId));

    if (product.getStock() < quantity) {
        throw new RuntimeException("Insufficient stock for product: " + product.getName());
    }

    product.setStock(product.getStock() - quantity);

    productRepository.save(product);
}
```

🔥 **Step 5: Now check in OrderStatus enum this are there or not**

```
public enum OrderStatus {

    CREATED,

    PAID,

    SHIPPED,

    DELIVERED,

    CANCELLED

}
```

🔥 Step 6: Update Order Creation

In placeOrder():

Change:

.orderStatus(OrderStatus.PLACED)

To:

.orderStatus(OrderStatus.CREATED)

Because payment has not happened yet.

🔥 Step 7: Update Payment Flow

In updatePaymentStatus():

Change this logic:

```
if (requestDto.getPaymentStatus() == PaymentStatus.PAID) {
   order.setOrderStatus(OrderStatus.CONFIRMED);
}
```

To:

```
if (requestDto.getPaymentStatus() == PaymentStatus.PAID) {
   order.setOrderStatus(OrderStatus.PAID);
}
```

Now lifecycle becomes correct:

CREATED → PAID

🔥 Step 8: Update Order Status

In order service service layer

OrderResponseDto updateOrderStatus(Long orderId, OrderStatus newStatus);

OrderServiceImpl :

```
@Transactional
public OrderResponseDto updateOrderStatus(Long orderId, OrderStatus newStatus) {

   Order order = orderRepository.findById(orderId)
         .orElseThrow(() ->
               new ResourceNotFoundException("Order not found with id: " + orderId));

   // Prevent invalid transitions
```

```
    if (order.getOrderStatus() == OrderStatus.CANCELLED ||
            order.getOrderStatus() == OrderStatus.DELIVERED) {
        throw new RuntimeException("Order status cannot be changed");
    }

    if (order.getOrderStatus() == OrderStatus.CREATED && newStatus != OrderStatus.PAID) {
        throw new RuntimeException("Order must be PAID first");
    }

    if (order.getOrderStatus() == OrderStatus.PAID && newStatus != OrderStatus.SHIPPED) {
        throw new RuntimeException("Order must be SHIPPED after PAID");
    }

    if (order.getOrderStatus() == OrderStatus.SHIPPED && newStatus != OrderStatus.DELIVERED) {
        throw new RuntimeException("Order must be DELIVERED after SHIPPED");
    }

    order.setOrderStatus(newStatus);
    orderRepository.save(order);

    return getOrderById(orderId);
}
```

🔥 **Step 9: In OrderController**

```
@PutMapping("/{orderId}/status")
public ResponseEntity<OrderResponseDto> updateOrderStatus(
        @PathVariable Long orderId,
        @RequestParam OrderStatus newStatus) {

    return ResponseEntity.ok(orderService.updateOrderStatus(orderId, newStatus));
}
```

🔥 **Step 9: Final Code for OrderServiceImpl**

```
package com.advann.order_service.service.serviceImpl;

import com.advann.order_service.client.CartClient;
import com.advann.order_service.client.ProductClient;
import com.advann.order_service.dto.*;
import com.advann.order_service.entity.Order;
import com.advann.order_service.entity.OrderItem;
import com.advann.order_service.enums.OrderStatus;
import com.advann.order_service.enums.PaymentStatus;
```

```
import com.advann.order_service.exception.ResourceNotFoundException;
import com.advann.order_service.payload.ApiResponse;
import com.advann.order_service.repository.OrderItemRepository;
import com.advann.order_service.repository.OrderRepository;
import com.advann.order_service.service.services.OrderService;
import org.springframework.transaction.annotation.Transactional;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.util.List;

@Service
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {

    private final OrderRepository orderRepository;
    private final OrderItemRepository orderItemRepository;
    private final CartClient cartClient;
    private final ProductClient productClient;
    private final ModelMapper modelMapper;

    @Override
    @Transactional
    public OrderResponseDto placeOrder(Long userId) {

        ApiResponse<CartResponseDto> cartResponse = cartClient.getCartByUserId(userId);

        if (cartResponse == null || cartResponse.getData() == null || cartResponse.getData().getItems().isEmpty()) {
            throw new ResourceNotFoundException("Cart is empty for userId: " + userId);
        }

        CartResponseDto cart = cartResponse.getData();

        // Create Order
        Order order = Order.builder()
                .userId(userId)
                .orderStatus(OrderStatus.CREATED)
                .paymentStatus(PaymentStatus.PENDING)
                .totalAmount(cart.getGrandTotal())
                .build();

        order = orderRepository.save(order);

        // Save Order Items
        for (CartItemResponseDto cartItem : cart.getItems()) {
```

```
        ApiResponse<ProductResponseDto> productResponse =
                productClient.getProductById(cartItem.getProductId());

        if (productResponse == null || productResponse.getData() == null) {
            throw new ResourceNotFoundException("Product not found with id: " + cartItem.getProductId());
        }

        ProductResponseDto product = productResponse.getData();

        if (product.getStock() < cartItem.getQuantity()) {
            throw new RuntimeException("Insufficient stock for product: " + product.getName());
        }

        // Reduce Stock in product-service
        productClient.reduceStock(cartItem.getProductId(), cartItem.getQuantity());

        OrderItem orderItem = OrderItem.builder()
                .order(order)
                .productId(cartItem.getProductId())
                .quantity(cartItem.getQuantity())
                .price(cartItem.getPrice())
                .build();

        orderItemRepository.save(orderItem);
    }

    // Clear cart after order placed
    cartClient.clearCart(userId);

    return getOrderById(order.getId());
}

@Override
public OrderResponseDto getOrderById(Long orderId) {

    Order order = orderRepository.findById(orderId)
            .orElseThrow(() -> new ResourceNotFoundException("Order not found with id: " + orderId));

    List<OrderItem> orderItems = orderItemRepository.findByOrderId(orderId);

    List<OrderItemResponseDto> responseItems = orderItems.stream()
            .map(item -> {

                ApiResponse<ProductResponseDto> productResponse =
                        productClient.getProductById(item.getProductId());
```

```
            ProductResponseDto product = productResponse.getData();

            OrderItemResponseDto dto = OrderItemResponseDto.builder()
                    .productId(item.getProductId())
                    .productName(product != null ? product.getName() : null)
                    .quantity(item.getQuantity())
                    .price(item.getPrice())
                    .totalPrice(item.getTotalPrice())
                    .build();

            return dto;
        })
        .toList();

    return OrderResponseDto.builder()
        .orderId(order.getId())
        .userId(order.getUserId())
        .orderStatus(order.getOrderStatus())
        .paymentStatus(order.getPaymentStatus())
        .totalAmount(order.getTotalAmount())
        .items(responseItems)
        .createdAt(order.getCreatedAt())
        .build();
}

@Override
public List<OrderResponseDto> getOrdersByUserId(Long userId) {

    List<Order> orders = orderRepository.findByUserId(userId);

    return orders.stream()
        .map(order -> getOrderById(order.getId()))
        .toList();
}

@Override
@Transactional
public OrderResponseDto cancelOrder(Long orderId) {

    Order order = orderRepository.findById(orderId)
        .orElseThrow(() ->
            new ResourceNotFoundException("Order not found with id: " + orderId));

    if (order.getOrderStatus() == OrderStatus.SHIPPED ||
        order.getOrderStatus() == OrderStatus.DELIVERED) {
        throw new RuntimeException("Order cannot be cancelled at this stage");
    }
```

```
        if (order.getOrderStatus() == OrderStatus.CANCELLED) {
            throw new RuntimeException("Order is already cancelled");
        }

        // Restore stock
        List<OrderItem> orderItems = orderItemRepository.findByOrderId(orderId);

        for (OrderItem item : orderItems) {
            productClient.increaseStock(item.getProductId(), item.getQuantity());
        }

        order.setOrderStatus(OrderStatus.CANCELLED);
        orderRepository.save(order);

        return getOrderById(orderId);
    }

    @Override
    public OrderResponseDto updatePaymentStatus(Long orderId, PaymentStatusUpdateRequestDto requestDto) {

    Order order = orderRepository.findById(orderId)
            .orElseThrow(() -> new ResourceNotFoundException("Order not found with id: " + orderId));

    order.setPaymentStatus(requestDto.getPaymentStatus());

    if (requestDto.getPaymentStatus() == PaymentStatus.PAID) {
        order.setOrderStatus(OrderStatus.PAID);
    }

    if (order.getOrderStatus() != OrderStatus.CREATED) {
        throw new RuntimeException("Payment already processed or invalid state");
    }

    orderRepository.save(order);

    return getOrderById(orderId);
}



    @Transactional
    public OrderResponseDto updateOrderStatus(Long orderId, OrderStatus newStatus) {

        Order order = orderRepository.findById(orderId)
                .orElseThrow(() ->
                        new ResourceNotFoundException("Order not found with id: " + orderId));
```

```
        // Prevent invalid transitions
        if (order.getOrderStatus() == OrderStatus.CANCELLED ||
                order.getOrderStatus() == OrderStatus.DELIVERED) {
            throw new RuntimeException("Order status cannot be changed");
        }

        if (order.getOrderStatus() == OrderStatus.CREATED && newStatus != OrderStatus.PAID) {
            throw new RuntimeException("Order must be PAID first");
        }

        if (order.getOrderStatus() == OrderStatus.PAID && newStatus != OrderStatus.SHIPPED) {
            throw new RuntimeException("Order must be SHIPPED after PAID");
        }

        if (order.getOrderStatus() == OrderStatus.SHIPPED && newStatus != OrderStatus.DELIVERED) {
            throw new RuntimeException("Order must be DELIVERED after SHIPPED");
        }

        order.setOrderStatus(newStatus);
        orderRepository.save(order);

        return getOrderById(orderId);
    }
}
```

🔥 **Step 10: We need to change in API-GATEWAY service JwtAuthFilter file also for ADMIN control**

```
package com.advann.api_gateway.security;

import lombok.RequiredArgsConstructor;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
@RequiredArgsConstructor
public class JwtAuthFilter implements GlobalFilter {

    private final JwtUtil jwtUtil;
```

```
private final WebClient.Builder webClientBuilder;

@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

    String path = exchange.getRequest().getURI().getPath();
    HttpMethod method = exchange.getRequest().getMethod();

    // =========================
    // PUBLIC ENDPOINTS
    // =========================
    if (path.startsWith("/api/auth/")
            || (path.startsWith("/api/products") && method == HttpMethod.GET)) {
        return chain.filter(exchange);
    }

    // =========================
    // AUTHORIZATION HEADER CHECK
    // =========================
    String authHeader = exchange.getRequest()
            .getHeaders()
            .getFirst(HttpHeaders.AUTHORIZATION);

    if (authHeader == null || !authHeader.startsWith("Bearer ")) {
        exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        return exchange.getResponse().setComplete();
    }

    String token = authHeader.substring(7);

    // =========================
    // TOKEN VALIDATION
    // =========================
    try {
        jwtUtil.validateToken(token);
    } catch (Exception e) {
        exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        return exchange.getResponse().setComplete();
    }

    // =========================
    // EXTRACT ROLE
    // =========================
    String role;
    try {
        role = jwtUtil.extractRole(token);
    } catch (Exception e) {
```

```
    exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
    return exchange.getResponse().setComplete();
}

// ========================
// PRODUCT WRITE → ADMIN ONLY
// ========================
if (path.startsWith("/api/products") && method != HttpMethod.GET) {
    if (!"ROLE_ADMIN".equals(role)) {
        exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
        return exchange.getResponse().setComplete();
    }
}

// ========================
// ORDER STATUS UPDATE → ADMIN ONLY
// Example: PUT /api/orders/5/status
// ========================
if (method == HttpMethod.PUT && path.matches("/api/orders/\\d+/status")) {
    if (!"ROLE_ADMIN".equals(role)) {
        exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
        return exchange.getResponse().setComplete();
    }
}

// ========================
// CART / ORDERS / PAYMENT → CUSTOMER OR ADMIN
// ========================
if (path.startsWith("/api/cart")
        || path.startsWith("/api/orders")
        || path.startsWith("/api/payments")) {

    if (!"ROLE_CUSTOMER".equals(role) && !"ROLE_ADMIN".equals(role)) {
        exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
        return exchange.getResponse().setComplete();
    }
}

// ========================
// BLACKLIST VALIDATION (USER SERVICE)
// ========================
return webClientBuilder.build()
        .get()
        .uri("http://USER-SERVICE/api/auth/validate-token")
        .header(HttpHeaders.AUTHORIZATION, "Bearer " + token)
        .retrieve()
        .bodyToMono(String.class)
```

```
            .flatMap(response -> chain.filter(exchange))
            .onErrorResume(e -> {
                exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
                return exchange.getResponse().setComplete();
            });
    }
}
```

🔥 **Step 11: We need to add this in ProductClient in order service**

```
@PutMapping("/api/products/increase-stock/{productId}")
ApiResponse<Void> increaseStock(
        @PathVariable Long productId,
        @RequestParam Integer quantity);
```

🔥 **Step 12: We need to update in product service service**

ProductService.java

```
void reduceStock(Long productId, Integer stock);
void increaseStock(Long productId, Integer quantity);
```

ProductServiceImpl.java

```
package com.advann.product_service.service.serviceImpl;

import com.advann.product_service.dto.PagedResponseDto;
import com.advann.product_service.dto.ProductImageResponseDto;
import com.advann.product_service.dto.ProductRequestDto;
import com.advann.product_service.dto.ProductResponseDto;
import com.advann.product_service.entity.Category;
import com.advann.product_service.entity.Product;
import com.advann.product_service.entity.ProductImage;
import com.advann.product_service.entity.SubCategory;
import com.advann.product_service.exceptions.InvalidFileException;
import com.advann.product_service.exceptions.ResourceNotFoundException;
import com.advann.product_service.repository.CategoryRepository;
import com.advann.product_service.repository.ProductImageRepository;
import com.advann.product_service.repository.ProductRepository;
import com.advann.product_service.repository.SubCategoryRepository;
import com.advann.product_service.service.services.ProductService;
import com.advann.product_service.service.services.S3Service;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import net.coobird.thumbnailator.Thumbnails;
import org.modelmapper.ModelMapper;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.*;
import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayOutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
public class ProductServiceImpl implements ProductService {

    private static final Logger log = LoggerFactory.getLogger(ProductServiceImpl.class);

    private final ProductRepository productRepository;
    private final ModelMapper modelMapper;
    private final CategoryRepository categoryRepository;
    private final SubCategoryRepository subCategoryRepository;
    private final ProductImageRepository productImageRepository;
    private final S3Service s3Service;

    @Value("${app.base-url}")
    private String baseUrl;

    private static final int MAX_GALLERY_IMAGES = 5;

    @Override
    public ProductResponseDto addProduct(ProductRequestDto productRequestDto) {

        log.info("Creating new product with name: {}", productRequestDto.getName());

        Product product = modelMapper.map(productRequestDto, Product.class);
        product.setId(null);

        Category category = categoryRepository.findById(productRequestDto.getCategoryId())
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Category not found with id: " + productRequestDto.getCategoryId()
                ));

        SubCategory subCategory = subCategoryRepository.findById(productRequestDto.getSubCategoryId())
                .orElseThrow(() -> new ResourceNotFoundException(
                        "SubCategory not found with id: " + productRequestDto.getSubCategoryId()
                ));
```

```
    if (!subCategory.getCategory().getId().equals(category.getId())) {
        throw new ResourceNotFoundException(
                "SubCategory id " + productRequestDto.getSubCategoryId() +
                        " does not belong to Category id " + productRequestDto.getCategoryId()
        );
    }

    product.setCategory(category);
    product.setSubCategory(subCategory);

    Product savedProduct = productRepository.save(product);

    log.info("Product created successfully with id: {}", savedProduct.getId());

    ProductResponseDto responseDto = modelMapper.map(savedProduct, ProductResponseDto.class);

    responseDto.setCategoryId(category.getId());
    responseDto.setCategoryName(category.getName());

    responseDto.setSubCategoryId(subCategory.getId());
    responseDto.setSubCategoryName(subCategory.getName());

    responseDto.setImageUrl(buildImageUrl(savedProduct.getImagePath()));

    return responseDto;
}

@Override
public List<ProductResponseDto> getAllProducts() {

    log.info("Fetching all products");

    List<Product> products = productRepository.findAll();

    log.info("Total products found: {}", products.size());

    return products.stream()
            .map(product -> {

                ProductResponseDto dto = modelMapper.map(product, ProductResponseDto.class);

                dto.setImageUrl(buildImageUrl(product.getImagePath()));

                if (product.getCategory() != null) {
                    dto.setCategoryId(product.getCategory().getId());
                    dto.setCategoryName(product.getCategory().getName());
                }
```

```
            if (product.getSubCategory() != null) {
                dto.setSubCategoryId(product.getSubCategory().getId());
                dto.setSubCategoryName(product.getSubCategory().getName());
            }

            return dto;
        })
        .collect(Collectors.toList());
}

@Override
public ProductResponseDto getProductById(Long id) {

    log.info("Fetching product by id: {}", id);

    Product product = productRepository.findById(id)
            .orElseThrow(() -> {
                log.warn("Product not found with id: {}", id);
                return new ResourceNotFoundException("Product not found with id: " + id);
            });

    ProductResponseDto dto = modelMapper.map(product, ProductResponseDto.class);

    dto.setImageUrl(buildImageUrl(product.getImagePath()));

    if (product.getCategory() != null) {
        dto.setCategoryId(product.getCategory().getId());
        dto.setCategoryName(product.getCategory().getName());
    }

    if (product.getSubCategory() != null) {
        dto.setSubCategoryId(product.getSubCategory().getId());
        dto.setSubCategoryName(product.getSubCategory().getName());
    }

    return dto;
}

@Override
public ProductResponseDto updateProduct(Long id, ProductRequestDto productRequestDto) {

    log.info("Updating product with id: {}", id);

    Product existing = productRepository.findById(id)
            .orElseThrow(() -> {
                log.warn("Cannot update. Product not found with id: {}", id);
```

```
            return new ResourceNotFoundException("Product not found with id: " + id);
        });

    existing.setName(productRequestDto.getName());
    existing.setPrice(productRequestDto.getPrice());
    existing.setStock(productRequestDto.getStock());

    Category category = categoryRepository.findById(productRequestDto.getCategoryId())
        .orElseThrow(() -> new ResourceNotFoundException(
            "Category not found with id: " + productRequestDto.getCategoryId()
        ));

    SubCategory subCategory = subCategoryRepository.findById(productRequestDto.getSubCategoryId())
        .orElseThrow(() -> new ResourceNotFoundException(
            "SubCategory not found with id: " + productRequestDto.getSubCategoryId()
        ));

    if (!subCategory.getCategory().getId().equals(category.getId())) {
        throw new ResourceNotFoundException(
            "SubCategory id " + productRequestDto.getSubCategoryId() +
                " does not belong to Category id " + productRequestDto.getCategoryId()
        );
    }

    existing.setCategory(category);
    existing.setSubCategory(subCategory);

    Product updated = productRepository.save(existing);

    ProductResponseDto responseDto = modelMapper.map(updated, ProductResponseDto.class);

    responseDto.setCategoryId(category.getId());
    responseDto.setCategoryName(category.getName());

    responseDto.setSubCategoryId(subCategory.getId());
    responseDto.setSubCategoryName(subCategory.getName());

    responseDto.setImageUrl(buildImageUrl(updated.getImagePath()));

    return responseDto;
}

@Override
public void deleteProduct(Long id) {

    log.info("Deleting product with id: {}", id);
```

```
        Product existing = productRepository.findById(id)
                .orElseThrow(() -> {
                    log.warn("Cannot delete. Product not found with id: {}", id);
                    return new ResourceNotFoundException("Product not found with id: " + id);
                });

        // ✅ delete primary image from S3
        deleteOldImageIfExists(existing);

        // ✅ delete gallery images from S3
        List<ProductImage> images = productImageRepository.findByProductId(existing.getId());

        for (ProductImage img : images) {
            if (img.getImagePath() != null) {
                s3Service.deleteFileByUrl(img.getImagePath());
            }
            if (img.getThumbnailPath() != null) {
                s3Service.deleteFileByUrl(img.getThumbnailPath());
            }
        }

        productRepository.delete(existing);

        log.info("Product deleted successfully with id: {}", id);
    }

    @Override
    public PagedResponseDto<ProductResponseDto> getAllProducts(
            int page,
            int size,
            String sortBy,
            String sortDir,
            String keyword,
            Long categoryId
    ) {

        Sort sort = sortDir.equalsIgnoreCase("desc")
                ? Sort.by(sortBy).descending()
                : Sort.by(sortBy).ascending();

        Pageable pageable = PageRequest.of(page, size, sort);

        Page<Product> productPage;

        if (categoryId != null && keyword != null && !keyword.isBlank()) {
            productPage = productRepository.findByCategoryIdAndNameContainingIgnoreCase(categoryId, keyword,
pageable);
```

```
    } else if (categoryId != null) {
        productPage = productRepository.findByCategoryId(categoryId, pageable);
    } else if (keyword != null && !keyword.isBlank()) {
        productPage = productRepository.findByNameContainingIgnoreCase(keyword, pageable);
    } else {
        productPage = productRepository.findAll(pageable);
    }

    List<ProductResponseDto> products = productPage.getContent()
            .stream()
            .map(product -> ProductResponseDto.builder()
                    .id(product.getId())
                    .name(product.getName())
                    .price(product.getPrice())
                    .stock(product.getStock())

                    .categoryId(product.getCategory() != null ? product.getCategory().getId() : null)
                    .categoryName(product.getCategory() != null ? product.getCategory().getName() : null)

                    .subCategoryId(product.getSubCategory() != null ? product.getSubCategory().getId() : null)
                    .subCategoryName(product.getSubCategory() != null ? product.getSubCategory().getName() : null)

                    .imagePath(product.getImagePath())
                    .imageUrl(buildImageUrl(product.getImagePath()))
                    .build()
            )
            .toList();

    return PagedResponseDto.<ProductResponseDto>builder()
            .content(products)
            .pageNumber(productPage.getNumber())
            .pageSize(productPage.getSize())
            .totalElements(productPage.getTotalElements())
            .totalPages(productPage.getTotalPages())
            .last(productPage.isLast())
            .build();
}

@Override
public ProductResponseDto uploadProductImage(Long productId, MultipartFile file) {

    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    validateImageFile(file);

    String imageUrl = s3Service.uploadFile(file, "products/full");
```

```
    product.setImagePath(imageUrl);

    Product savedProduct = productRepository.save(product);

    ProductResponseDto dto = modelMapper.map(savedProduct, ProductResponseDto.class);
    dto.setImageUrl(savedProduct.getImagePath());

    return dto;
}

private void validateImageFile(MultipartFile file) {

    if (file == null || file.isEmpty()) {
        throw new InvalidFileException("File is required. Please upload a file.");
    }

    String contentType = file.getContentType();

    if (contentType == null ||
            !(contentType.equals("image/jpeg") ||
                    contentType.equals("image/png"))) {
        throw new InvalidFileException("Only JPG and PNG images are allowed");
    }

    long maxSize = 2 * 1024 * 1024;
    if (file.getSize() > maxSize) {
        throw new InvalidFileException("File size must be less than 2MB");
    }
}

private void deleteOldImageIfExists(Product product) {

    if (product.getImagePath() == null || product.getImagePath().isBlank()) {
        log.info("No primary image found for product id: {}", product.getId());
        return;
    }

    try {
        s3Service.deleteFileByUrl(product.getImagePath());
        log.info("Old primary image deleted from S3 for product id: {}", product.getId());
    } catch (Exception e) {
        log.warn("Failed to delete old image from S3. Skipping delete. {}", e.getMessage());
    }
}

private String buildImageUrl(String imagePath) {
```

```
    if (imagePath == null || imagePath.isBlank()) {
        return baseUrl + "/images/default.jpg";
    }

    // ✅ S3 imagePath already contains full URL
    return imagePath;
}

@Override
public ProductResponseDto deleteProductImage(Long productId) {

    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    if (product.getImagePath() == null || product.getImagePath().isBlank()) {
        throw new InvalidFileException("No image found for this product.");
    }

    // ✅ delete from S3
    s3Service.deleteFileByUrl(product.getImagePath());

    product.setImagePath(null);

    Product savedProduct = productRepository.save(product);

    ProductResponseDto dto = modelMapper.map(savedProduct, ProductResponseDto.class);
    dto.setImageUrl(null);

    return dto;
}

@Override
public ProductResponseDto updateProductImage(Long productId, MultipartFile file) {

    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    validateImageFile(file);

    // ✅ delete old primary image from S3
    deleteOldImageIfExists(product);

    // ✅ upload new image to S3
    String imageUrl = s3Service.uploadFile(file, "products/full");

    product.setImagePath(imageUrl);
```

```
    Product savedProduct = productRepository.save(product);

    ProductResponseDto dto = modelMapper.map(savedProduct, ProductResponseDto.class);
    dto.setImageUrl(savedProduct.getImagePath());

    return dto;
}

@Override
public List<ProductImageResponseDto> uploadProductImages(Long productId, List<MultipartFile> files) {

    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    if (files == null || files.isEmpty()) {
        throw new InvalidFileException("Please upload at least one image.");
    }

    long existingCount = productImageRepository.countByProductId(productId);

    if (existingCount + files.size() > MAX_GALLERY_IMAGES) {
        throw new InvalidFileException(
                "Maximum " + MAX_GALLERY_IMAGES + " images allowed per product. Already uploaded: "
                        + existingCount + ", trying to upload: " + files.size()
        );
    }

    List<ProductImageResponseDto> responseList = new ArrayList<>();

    for (MultipartFile file : files) {

        validateImageFile(file);

        try {
            ByteArrayOutputStream fullOutputStream = new ByteArrayOutputStream();

            Thumbnails.of(file.getInputStream())
                    .size(800, 800)
                    .outputQuality(0.8)
                    .toOutputStream(fullOutputStream);

            byte[] fullBytes = fullOutputStream.toByteArray();

            ByteArrayOutputStream thumbOutputStream = new ByteArrayOutputStream();

            Thumbnails.of(file.getInputStream())
```

```
                .size(300, 300)
                .outputQuality(0.7)
                .toOutputStream(thumbOutputStream);

            byte[] thumbBytes = thumbOutputStream.toByteArray();

            String fullImageUrl = s3Service.uploadBytes(fullBytes, file.getContentType(), "products/gallery/full");
            String thumbImageUrl = s3Service.uploadBytes(thumbBytes, file.getContentType(),
"products/gallery/thumb");

            ProductImage productImage = ProductImage.builder()
                .imagePath(fullImageUrl)
                .thumbnailPath(thumbImageUrl)
                .product(product)
                .build();

            ProductImage savedImage = productImageRepository.save(productImage);

            responseList.add(ProductImageResponseDto.builder()
                .id(savedImage.getId())
                .imagePath(savedImage.getImagePath())
                .imageUrl(savedImage.getImagePath())
                .thumbnailPath(savedImage.getThumbnailPath())
                .thumbnailUrl(savedImage.getThumbnailPath())
                .build());

        } catch (Exception e) {
            throw new InvalidFileException("Failed to upload image to S3.");
        }
    }

    if (!responseList.isEmpty()) {
        ProductImageResponseDto latestUploadedImage = responseList.get(responseList.size() - 1);

        product.setImagePath(latestUploadedImage.getImagePath());
        productRepository.save(product);

        log.info("Primary image updated to latest uploaded image for product id: {}", product.getId());
    }

    return responseList;
}

@Override
public List<ProductImageResponseDto> getProductImages(Long productId) {

    productRepository.findById(productId)
```

```
        .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    List<ProductImage> images = productImageRepository.findByProductId(productId);

    return images.stream()
            .map(img -> ProductImageResponseDto.builder()
                    .id(img.getId())
                    .imagePath(img.getImagePath())
                    .imageUrl(img.getImagePath())
                    .thumbnailPath(img.getThumbnailPath())
                    .thumbnailUrl(img.getThumbnailPath())
                    .build())
            .toList();
}

@Override
public void deleteProductImageById(Long imageId) {

    ProductImage productImage = productImageRepository.findById(imageId)
            .orElseThrow(() -> new ResourceNotFoundException("Image not found with id: " + imageId));

    Product product = productImage.getProduct();

    boolean isPrimary = product.getImagePath() != null &&
            product.getImagePath().equals(productImage.getImagePath());

    // ✅ delete full image from S3
    if (productImage.getImagePath() != null) {
        s3Service.deleteFileByUrl(productImage.getImagePath());
    }

    // ✅ delete thumbnail from S3
    if (productImage.getThumbnailPath() != null) {
        s3Service.deleteFileByUrl(productImage.getThumbnailPath());
    }

    productImageRepository.delete(productImage);
    log.info("Gallery image deleted from DB with id: {}", imageId);

    if (isPrimary) {

        List<ProductImage> remainingImages =
                productImageRepository.findByProductIdOrderByCreatedAtDesc(product.getId());

        if (!remainingImages.isEmpty()) {
            product.setImagePath(remainingImages.get(0).getImagePath());
            log.info("Primary image updated to another gallery image for product id: {}", product.getId());
```

```
        } else {
            product.setImagePath(null);
            log.info("No gallery images left. Primary image reset to null for product id: {}", product.getId());
        }

        productRepository.save(product);
    }
}

@Override
public ProductResponseDto setPrimaryProductImage(Long imageId) {

    ProductImage productImage = productImageRepository.findById(imageId)
            .orElseThrow(() -> new ResourceNotFoundException("Image not found with id: " + imageId));

    Product product = productImage.getProduct();

    product.setImagePath(productImage.getImagePath());

    Product savedProduct = productRepository.save(product);

    ProductResponseDto dto = modelMapper.map(savedProduct, ProductResponseDto.class);
    dto.setImageUrl(savedProduct.getImagePath());

    return dto;
}

@Override
public PagedResponseDto<ProductImageResponseDto> getProductImagesWithPagination(Long productId, int
page, int size, String sortDir) {

    productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found with id: " + productId));

    Sort sort = sortDir.equalsIgnoreCase("desc")
            ? Sort.by("createdAt").descending()
            : Sort.by("createdAt").ascending();

    Pageable pageable = PageRequest.of(page, size, sort);

    Page<ProductImage> imagePage = productImageRepository.findByProductId(productId, pageable);

    List<ProductImageResponseDto> images = imagePage.getContent()
            .stream()
            .map(img -> ProductImageResponseDto.builder()
                    .id(img.getId())
                    .imagePath(img.getImagePath())
```

```
                .imageUrl(img.getImagePath())
                .thumbnailPath(img.getThumbnailPath())
                .thumbnailUrl(img.getThumbnailPath())
                .build())
        .toList();

    return PagedResponseDto.<ProductImageResponseDto>builder()
        .content(images)
        .pageNumber(imagePage.getNumber())
        .pageSize(imagePage.getSize())
        .totalElements(imagePage.getTotalElements())
        .totalPages(imagePage.getTotalPages())
        .last(imagePage.isLast())
        .build();
}

@Override
@Transactional
public void reduceStock(Long productId, Integer quantity) {

    if (quantity <= 0) {
        throw new RuntimeException("Quantity must be greater than 0");
    }

    Product product = productRepository.findById(productId)
        .orElseThrow(() ->
            new ResourceNotFoundException("Product not found with id: " + productId));

    if (product.getStock() < quantity) {
        throw new RuntimeException("Insufficient stock for product: " + product.getName());
    }

    product.setStock(product.getStock() - quantity);

    productRepository.save(product);
}

@Override
@Transactional
public void increaseStock(Long productId, Integer quantity) {

    if (quantity <= 0) {
        throw new RuntimeException("Quantity must be greater than 0");
    }

    Product product = productRepository.findById(productId)
        .orElseThrow(() ->
```

```
                new ResourceNotFoundException("Product not found with id: " + productId));

        product.setStock(product.getStock() + quantity);

        productRepository.save(product);
    }
}
```

ProductController.java

```
@PutMapping("/reduce-stock/{id}/{stock}")
public ResponseEntity<ApiResponse<Object>> reduceStock(
        @PathVariable Long id,
        @PathVariable Integer stock
) {

    productService.reduceStock(id, stock);

    ProductResponseDto product = productService.getProductById(id);

    return ResponseEntity.ok(
            ApiResponse.builder()
                    .success(true)
                    .message("Stock reduced successfully")
                    .data(product)
                    .build()
    );
}

@PutMapping("/increase-stock/{productId}")
public ResponseEntity<ApiResponse<Void>> increaseStock(
        @PathVariable Long productId,
        @RequestParam Integer quantity) {

    productService.increaseStock(productId, quantity);

    return ResponseEntity.ok(
            new ApiResponse<>(true, "Stock increased successfully", null)
    );
}
```

🔥 **Step Final: We need small update here also**

```
package com.advann.product_service.dto;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import lombok.Data;

import java.math.BigDecimal;

@Data
public class ProductRequestDto {
    @NotBlank(message = "Product name cannot be blank")
    private String name;

    @NotNull(message = "Price cannot be null")
    //inclusive = false means the minimum value is not allowed, so the number must be strictly greater than the given value (ex: price must be > 0.0).
    @DecimalMin(value = "0.0", inclusive = false, message = "Price should be greater than 0")
    private BigDecimal price;

    @NotNull(message = "stock cannot be null")
    @Min(value = 1, message = "stock must be at least 1")
    private Integer stock;

    @NotNull(message = "Category Id is required")
    private Long categoryId;

    @NotNull(message = "SubCategory Id is required")
    private Long subCategoryId;
}
```

Done.