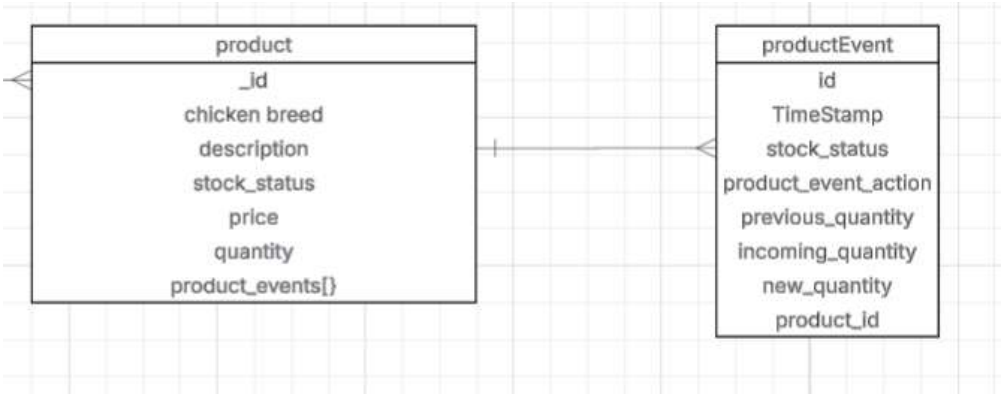
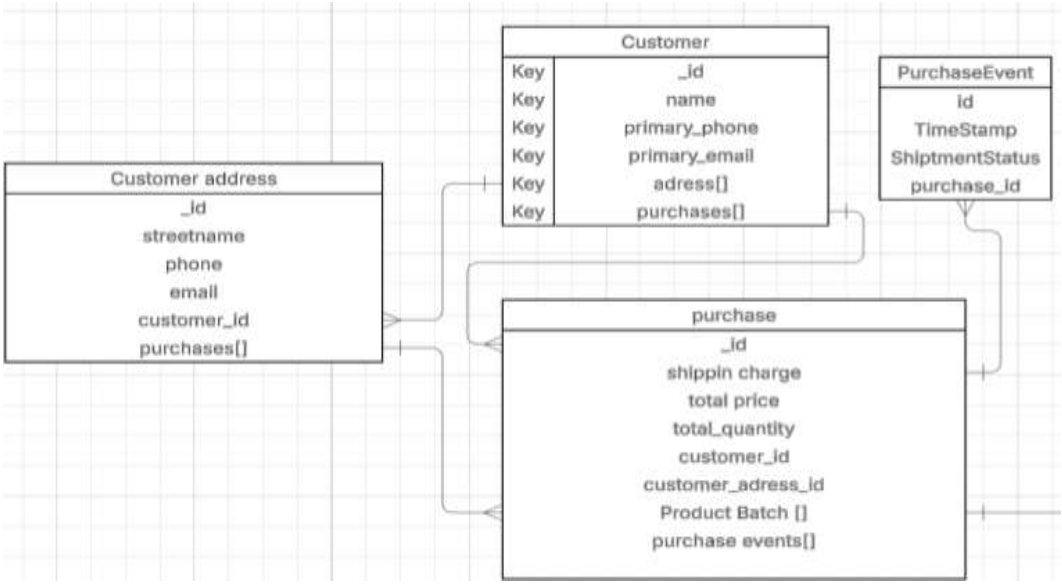


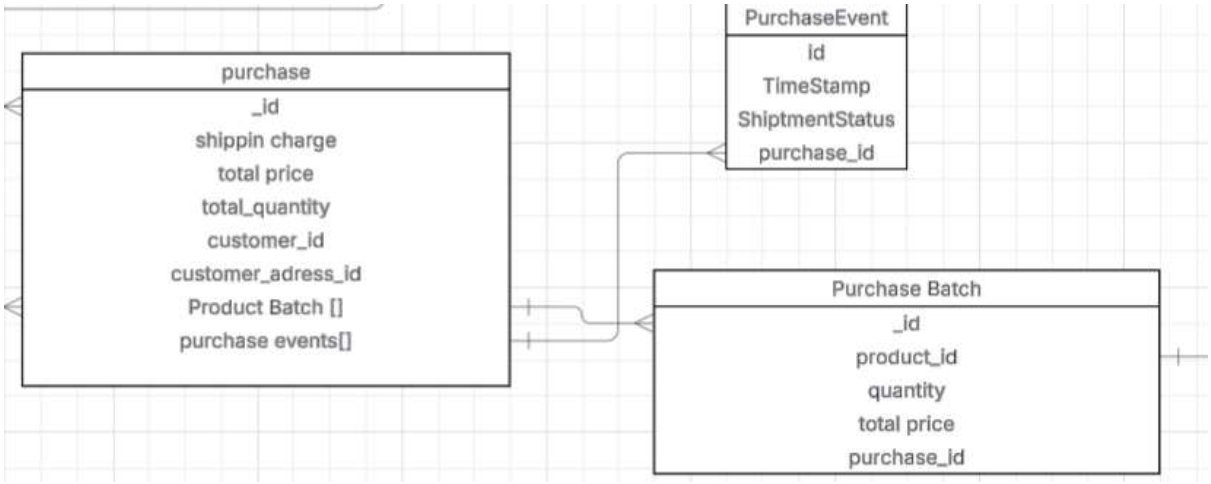
Product:

- Name = Breed
- Status tracked via stock_status and productEvent [] table
- Events created on purchase, cancel, or restock



Customer / CustomerAddress:

- Both have phone and email to allow contact per address independently
- Order history tracked via Purchases [] and PurchaseEvent [] tables
- Both Customer and CustomerAddress link to Purchase(order) for data integrity (if address is deleted, purchase still traces to customer)



Order = Purchase

- List of Products = PurchaseBatch tracks what kind of product (BLACK, GOLDEN, WHITE, BROWN), quantity of batch, and batch price
- Total quantity of all batches in Purchase
- PurchaseEvent automatically created on purchase with shipping status NOT_SHIPPED.
- Links to Customer and CustomerAddress (shipping address)

We use DTO Records for response entities rather than @JsonIgnore annotations. This provides endpoint-specific control over field exposure, avoiding scenarios where a field must be ignored in one response but included in another. DTOs also enable us to customize field names for improved API readability without altering entity definitions. Importantly, DTOs act as an explicit whitelist, ensuring only intended fields are exposed, preventing accidental data leakage if new entity fields are added without corresponding @JsonIgnore annotations.

NB: Please run <localhost:8080/api/test/init> on postman as a get request for test data.
- We have postman collection in src folder intelliji.

1. Fetching a customer should show their addresses and order history
Json Payload via Postman customer/3
ThymeLeaf customer/3

2. Fetching a Purchase should show the customer and shipping address.
Json Payload via Postman purchase/2
ThymeLeaf purchase/2
Purchase from a Special man ←← Please click to see a handsome chicken

3. Placing an order should update the status and quantity on hand of a product, and the system should not allow products to be ordered that are out of stock.

- Successful Purchase → Postman Collection → <Create Purchase>
- Out of Stock Purchase → Postman Collection →< Create purchase not enough stock>

PurchaseOrchestrationService.create() calls ProductOrchestrationService.decreaseStock() to update stock. Stock validation ensures product.quantity >= order.quantity (prevents overselling). Each change tracked in ProductEvent (previous, incoming, new quantities).

EXCEPTION:
Custom exceptions with descriptive names handled by a global error handler that maps exceptions to HTTP responses. Logging implemented for business logic, mapping, and debugging.

TESTING:
Business logic isolated in PurchaseOrchestrationService and ProductOrchestrationService (other service classes handle only CRUD). This prevents circular dependencies without @Lazy and centralizes logic. Testing focuses on orchestration layers where business rules are enforced.

JACOCO: 73% coverage (all classes) | 95% coverage (excluding DTOs, mappers, test data, controllers)

DESIGN:
We considered consolidating PurchaseEventService and PurchaseBatchService into PurchaseService, and ProductEventService into ProductService, since they belong to the same domain. However, to comply with the requirement that "services should not talk to repos other than their own," we created separate service classes. We did consolidate related services under single controllers (e.g., PurchaseController handles purchase, batch, and event endpoints).